# ADAPTIVE UNSTRUCTURED SPACETIME MESHING FOR FOUR-DIMENSIONAL SPACETIME DISCONTINUOUS GALERKIN FINITE ELEMENT METHODS

BY

ALEXANDER DAVID MONT

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Professor Jeff Erickson

# Abstract

We describe the spacetime discontinuous Galerkin method, a new type of finite-element method which promises dramatic improvement in solution speed for hyperbolic problems. These methods require the generation of spacetime meshes that satisfy a special *causality constraint*. This work focuses on the extension of the existing $2d\times$time spacetime meshing algorithm known as TentPitcher to $3d\times$time problems. We review existing work based on TentPitcher. Then, we extend TentPitcher to $3d\times$time and derive methods for handling mesh adaptivity operations. Next, we describe the software we have developed to implement our algorithms and give preliminary results of testing. We also identify unresolved theoretical and engineering issues associated with our new methods and suggest directions for further research.

# Acknowledgements

This project would not have been possible without the support of many people. In particular, I would like to thank my advisor Jeff Erickson, who guided me through many aspects of our research project, including teaching me how to generate high-quality visualizations, follow good software development practices, and write clear, concise, and correct proofs. Most importantly, Professor Erickson convinced me that an M.S. degree was a better fit for me than a Ph.D. - and I have no doubts now that the decision to switch from a Ph.D. program to the M.S. program was the right choice. Also, Professor Erickson looked at many early drafts of this thesis and provided valuable contributions. I would also like to thank the other members of my research team: Robert Haber, Reza Abedi, and Raj Kumar Pal, who worked on developing the finite element solver parts of our software. I would also like to thank Rhonda McElroy of the Computer Science Department for providing valuable advice and guidance during my time at the University of Illinois at Urbana-Champaign, and Mary Beth Kelley for guiding me through the thesis submission process.

# Table of Contents

# Chapter 1

# Introduction

## 1.1 Overview

This thesis is based on a multi-year effort to develop software for spacetime discontinuous Galerkin finite element methods [30, 43], a new class of finite element methods that promise dramatically faster solutions for hyperbolic partial differential equations. Spacetime discontinuous Galerkin methods rely on a mesh of the spacetime analysis domain, and they require that the boundaries between elements in this mesh satisfy a special *cone constraint* [30, 58]. We use an algorithm known as TentPitcher [58] to build a mesh of the spacetime domain that satisfies this condition. This thesis provides an overview of the software and its functionality. It also includes new theoretical contributions in the area of adaptive spacetime meshing. This thesis is organized as follows. First, we provide an overview of the principles, history, and development of spacetime discontinuous Galerkin methods prior to this work. Second, we describe the general principles of the $3d\times$time spacetime discontinuous Galerkin method and give an overview of the architecture and functionality of our SDG software. Third, we describe new theoretical developments needed to extend spacetime discontinuous Galerkin methods from $2d\times$time problems such as those treated by Abedi *et al.* [2] to 3d×time problems. Fourth, we describe the post-processing and output features of our software and provide examples of its output.

## 1.2 Terminology

A *simplex* is the generalization of a tetrahedron to an arbitrary number of dimensions [40]. Formally, a $d$-dimensional simplex, or $d$-simplex, is the convex hull of a set of $d+1$ affinely independent points, called the *vertex set* of the simplex. A *face* of a simplex $S$ is a simplex whose vertex set is a subset of the vertex set of $S$. A *facet* of $S$ is a face of $S$ with dimension exactly one less than the dimension of $S$. A *coface* of $S$ is a simplex whose vertex set contains the vertex set of $S$. A *cofacet* of $S$ is a coface of $S$ with dimension exactly one greater than the dimension of $S$.

A collection $D$ of simplices is a simplicial complex [40] if all faces of a simplex in $D$ are themselves in $D$ and the intersection of any two simplices in $D$ is a face of both of the simplices in $D$.

The *dimension* of a simplicial complex is the dimension of the highest-dimensional cells in the complex. A *facet* of a simplicial complex is a cell whose dimension is equal to the dimension of the complex.

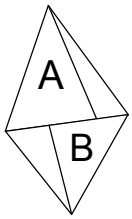


**Figure 1.1.** This mesh is not a simplicial complex, because the intersection of the triangles $A$ and $B$ is not a facet of either.

## 1.3 Main Principles

Finite element methods are an important class of methods used in science and engineering to simulate physical phenomena by solving partial differential equations (PDEs). A comprehensive overview of finite element methods and applications is given by Zienkiewicz and Taylor [63, 62, 61]. In such problems, the spatial region under analysis is divided into a large number of simpler volumes such as triangles, quadrilaterals, tetrahedra, or hexahedra. These simpler volumes are known as *cells*, and the collection of volumes is known as a *mesh*. Many methods of generating a variety of mesh types for finite element methods are known [56, 21, 27, 14]. A *simplicial mesh* is a mesh that is a simplicial complex. The most common types of simplicial meshes used in finite element applications are triangular and tetrahedral meshes. Many methods of generating triangular and tetrahedral methods are known; a survey of such methods is given by Owen [42]. Finite element methods represent the solution to the PDE as a linear combination of basis functions, each with support over one or a small number of cells. These methods transform the PDE into a system of linear equations whose solution gives coefficients of the basis functions that approximate the solution to the PDE.

In many applications, the system to be modeled changes over time. In this case, most solution methods use a discrete time-marching scheme. In these methods, generally a specific *time step* $\Delta t$ is chosen, and the entire mesh is solved at time 0, then at time $\Delta t$, then at time $2\Delta t$, and so on. An overview of a variety of different time-marching schemes is given by Wood [59], and an example of a time marching scheme being used to solve a particular problem is given by Jameson *et al.* [23]. However, these methods can be computationally expensive because a potentially very large system of linear equations must be solved at each time step.

An important class of PDEs are known as *hyperbolic* PDEs. Hyperbolic PDEs generally model wave-like behavior, such as the propagation of shockwaves or sound waves. These physical phenomena have a finite *maximum wave speed*, which defines the maximum speed at which disturbances can propagate. For linear hyperbolic equations with homogeneous material properties, this maximum wave speed is the same everywhere. Our work assumes constant maximum wave speeds.

The spacetime discontinuous Galerkin methods developed by Haber and others [4, 7, 8, 6, 35, 3, 44, 34, 43] do not use a fixed time-marching scheme, but rather generates a mesh of the entire *spacetime* analysis domain $\Omega \times [0, t]$, where $\Omega$ is the spatial domain and $[0, t]$ is the time interval over which the function is to be solved. We refer to the time value as height, so, for instance, "above" means at a higher time value. As explained below, we construct the spacetime mesh in such a way that the cells are divided into small *patches*, that are solved in sequence. The solution in each patch depends only on initial conditions and on previously solved patches, so each patch can be solved separately. This feature enables easier parallelism; it also enables improved solution speed because each system of linear equations is much smaller.

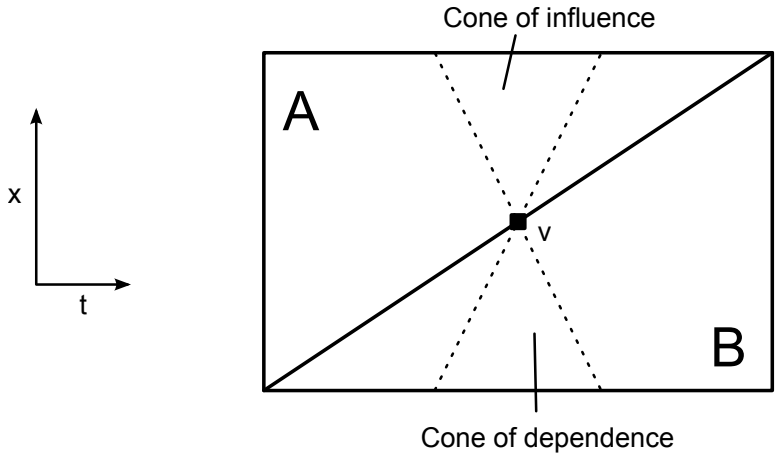


**Figure 1.2.** The solution in cell A cannot influence the solution in cell B.

Consider the $1d\times$time mesh shown in Figure 1.2. Any given point in the spacetime domain has a *cone of influence*, which indicates the set of spacetime points that depend on the solution at the given point, and a *cone of dependence*, which indicates the set of spacetime points that can influence the solution at the given point. The slopes of the sides of these cones are equal to the inverse of the maximum wave speed. If the line segment that separates two cells has slope less than the slope of the cones, no point in the lower cell is in the cone of influence of any point in the higher cell. Thus, cell $B$ can be solved before cell $A$, and we say that this line segment is *causal*.

These causality conditions generalize easily to higher dimensions. Consider a simulation over a $d$-dimensional spatial domain, so the highest-dimensional spacetime cells are $d+1$-dimensional. Consider a $d$-dimensional facet $S$ which separates two of these cells. If the slope of $S$ is below the inverse wave speed, then the facet is causal.

Given any two spacetime cells, we say that $A$ can *influence* $B$ if there is a point in $A$ that is inside the cone of dependence of a point in $B$. If two $d+1$-dimensional cells $A$ and $B$ share a $d$-dimensional noncausal facet, then they can both influence each other. If two $d+1$-dimensional cells $A$ and $B$ share a $d$-dimensional causal facet, then the one below the facet can influence the one above the facet, but not vice versa. The "can influence" relation defines a directed graph over the cells. Each strongly connected component of this graph is *coupled* [30, 58], and all the cells in such a component must be solved together.

TentPitcher creates a spacetime mesh divided into patches, where each patch contains some cells marked as *active* and others as *inactive*. (The same cell may be active in one patch and inactive in another patch.) Within each patch, all the active cells may be coupled. However, the patches are partially ordered by causality; i.e. they can be solved in sequence such that every patch is solved after all the patches they are influenced by. When we solve a patch $A$, we compute the solution data for all the active cells of $A$. To solve a patch $A$, we need the initial conditions, which are defined over the lower boundary of the patch. By construction, each cell in this lower boundary either is in the lower boundary of the spacetime mesh, in which case the initial conditions are known from the initial conditions of the problem, or it was an active cell of a previously solved patch, in which case the solution for that cell has already been computed. TentPitcher creates a spacetime mesh that satisfies these conditions patch by patch.

The general TentPitcher algorithm, first formulated by Üngör and Sheffer [58], generates an unstruc-

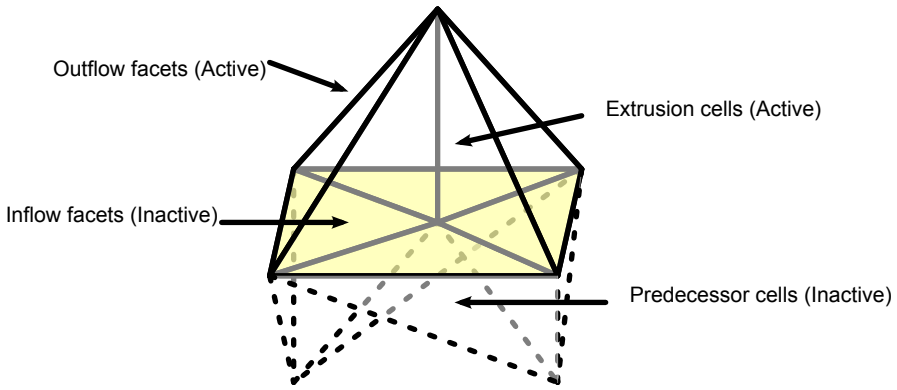


**Figure 1.3.** The parts of a patch.

tured mesh of the entire spacetime domain using an advancing-front scheme. The input to TentPitcher is a simplicial mesh $M$ of some $d$-dimensional spatial domain $\Omega$. TentPitcher assigns a *time coordinate* to each vertex of $M$. We also define a *time function* over the mesh, such that for each cell, the restriction of the time function to that cell is the linear interpolation of the time coordinates of that cell's vertices. The graph of this function is known as the *front*. By construction, the front is piecewise linear, with one piece for each cell in the space mesh. We say that a cell is *causal* if and only if the gradient of the time function on that cell is less than the inverse of the maximum wave speed. In our work, we choose units so that the maximum wave speed is 1.

TentPitcher generates a series of patches. To generate a patch, we first select a vertex $x$ in $M$ with a time value that is a local minimum. Then, we select a *pitch height* $h$. We must set $h$ high enough so that when the time value of $x$ is increased by $h$, $x$ is no longer a local minimum. We must also set $h$ low enough that when the time value of $x$ is increased by $h$, all the facets of $M$ are still causal.

To create a new patch, we generate a *new vertex*, $x'$ at the same spatial location of $x$ but with time value $t(x) + h$, where $t(x)$ is the time value of $x$. Each cell $C$ in the spatial domain that contains $x$ is considered an *inflow facet* of the new patch. For each inflow facet, we also generate an *extrusion cell* of dimension one higher than that of $C$, which is the convex hull of $C$ and $x'$. Each extrusion cell has an *outflow facet* of dimension equal to that of $C$, which contains $x'$ and all the vertices of $C$ except $x$. These new cells will become part of the patch. The extrusion cells and outflow facets are considered active cells of the new patch. By construction, the inflow and outflow facets of the new patch are causal, which ensures that the active cells are coupled only to other active cells in the same patch.

The inflow facets and their preceding cofacets are considered part of the patch. These predecessors are included because some solution methods only solve for solutions on the top-level spacetime cells, so the solution data is in fact in the predecessor cells rather than the inflow facets.

After solving a patch, we update the front by increasing the time value of the vertex pitched by $h$, so the graph of the time function agrees with the new front. Equivalently, to update the front, we replace the inflow facets of the patch with the outflow facets of the patch.

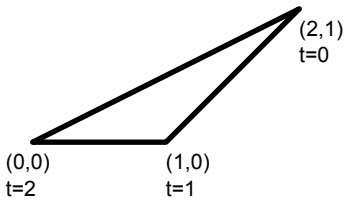


**Figure 1.4.** An example of how TentPitcher can get stuck. $\nabla\tau(S, \mathbf{t}) = (-1, 0)$ and so the triangle is causal, but any pitching of the lowest vertex will make the triangle noncausal.

## 1.4 Existing Work

To generate a spacetime mesh of the entire analysis domain, TentPitcher must eventually pitch every vertex arbitrarily far forward in time. Less formally, we must ensure that TentPitcher never "gets stuck." We must carefully choose $h$ - the height of each tent - at each step. Üngör and Sheffer [58] show that with a $2d$ spatial domain, if all the angles of each triangle in the space mesh are acute, it suffices to pitch every vertex to the highest value such that all the incident triangles remain causal.

However, if the input mesh contains obtuse angles, TentPitcher can get stuck (see Figure 1.4) [17]. To avoid this problem, we impose an additional set of *progress constraints*, which define whether a simplex in the front is *valid*. The progress constraints are formulated so that any valid facet is also causal, and if all the facets of the front are valid then the front can be advanced arbitrarily far forward in time and remain valid. Erickson *et al.* [17] describe progress constraints that ensure progress for an arbitrary number of dimensions, regardless of the angles in the space mesh.

Finite element simulations can often be improved through the use of adaptive remeshing [46]. In adaptive refinement, the mesh changes over the course of the simulation, becoming finer in areas where the solution changes rapidly and coarser in areas where the solution does not change rapidly. Adaptive remeshing enables computational effort to be concentrated in the most important areas, significantly reducing computation time needed for accuracy [1]. Abedi *et al.* [2] developed a method of incorporating adaptive refinement and coarsening into the $2d$ version of TentPitcher. Whenever a patch is solved, an estimate of the numerical error is computed along with the solution. If the error is above a given threshold, then the patch is rejected, the front is not updated, and some of the cells are refined instead. If the error is below a given threshold, then some of the new facets of the front are marked as coarsenable, and these elements become coarsened when such coarsening is possible.

The key challenge of adaptive refinement is that we must refine on demand; if all the cells in the mesh are valid, they must still be valid after refinement. Abedi *et al.* [2] perform adaptive $2d\times$time meshing using the *newest-vertex bisection* method of Sewell [47] and Mitchell [36, 37, 38]. This method has the property that the descendants of any triangle lie in exactly eight different homothety classes, and Abedi *et al.* use this fact to generate a working set of progress constraints. Abedi *et al.* also explain how to coarsen by undoing previous refinements; coarsening does not require separate progress constraints because coarsening is not required to succeed every time. If a coarsening operation would cause the new mesh to contain invalid cells, that operation fails.

Thite [54] additionally extended the $2d\times$time TentPitcher algorithm to support nonlinear problems, in which the maximum wave speed can vary depending on the solution [55]. This variation poses particular challenges for TentPitcher because the cone constraint depends on non-local information. Thite also

extended TentPitcher to support smoothing operations, where vertices can move in space as well as time when they are pitched [54]. Thite used smoothing operations because smoothing has been previously shown to improve mesh quality in a variety of settings, such as by Knupp [25] and Freitag and Ollivier-Gooch [20].

Zhou, Garland, and Haber [60] developed a method of visualizing the results of an SDG solution. Zhou *et al.*'s method takes a series of time-slices of the spacetime mesh, so each time slice consists of triangles and quadrilaterals. The SDG solution is used to compute two scalar fields: one of which is mapped to color, and one of which is mapped to height. The height field is used to deform the surface, and Zhou *et al.*'s software computes this deformation on a per-pixel basis.

The SDG solution and visualization software has been used to solve a wide variety of problems, including linearized elastodynamics and fracture models by Abedi *et al.* [4, 7, 8, 6] and Miller *et al.* [35], evolving discontinuities and shock capturing by Abedi *et al.* [3] and Palaniappan *et al.* [44], heat conduction by Miller and Haber [34], and scalar conservation laws by Palaniappan *et al.* [43].

## 1.5 Our Contributions

This thesis extends the work described above in several ways. Our main theoretical contribution is a set of *progress constraints* that enable on-demand refinement in 3 dimensions. First, we describe a set of generic progress criteria that ensure that TentPitcher will reach an arbitrarily high time value. Next, we describe a $3d$ refinement algorithm called Bänsch refinement [11, 12], which is a 3D analogue of newest-vertex bisection. As with newest-vertex bisection, the descendants of each tetrahedron fall into a finite number of homothety classes, and we use this fact to obtain appropriate progress constraints.

Generalizing earlier work in $2d\times$time, we also implement other mesh improvement operations in $3d$. These operations include coarsening by edge contraction, which has been shown by Cignoni *et al.* [15] and Ollivier-Gooch [41] to reduce the number of vertices in a mesh while maintaining mesh quality. Another such operation is edge flipping. Edge flipping is a key operation in Delaunay refinement, a technique which has been shown by Shewchuk to produce good-quality triangular and tetrahedral meshes [48, 49]. Edge flipping has also been shown by Klingner and Shewchuk [24] to be able to dramatically improve the quality of an existing mesh. A third such operation is smoothing by vertex movement, which is ideal for our purposes because it does not change the topology of the mesh. There are several methods for determining the ideal location to move each vertex. For instance, Laplacian smoothing moves each vertex to the centroid of its neighbors if possible [22, 18, 19]. Other methods optimize a quality metric, such as the methods of Knupp [25], Amenta *et al.* [9], and Parthasarathy and Kodiyalam [45]. We show how to integrate these operations with refinement by remarking some of the tetrahedra that were changed, in order to make the set of markings legal. We emphasize that although this method appears to work in practice, we do not have a proof that this method always leaves the mesh in a state where arbitrary further refinement is possible.

The rest of this thesis is not as theoretically novel, but does involve significant software engineering challenges. First, we implemented a modular software architecture that enables the finite element solution, mesh generation, pre-processing, and post-processing components of the software to be changed independently of each other. The same processing pipeline can be used regardless of the dimension of the space mesh. Next, we implemented a tool that enables visualization of the solution data. Similarly to Zhou *et al.*'s work [60], our tool enables the visualization of time-slices of the solution data, and displays two scalar fields simultane-

ously: one mapped to color and the other mapped to height. The height field is used to deform the displayed surface in a normal direction. While our visualization tool does not yet offer the same pixel-exact rendering capabilities as those of Zhou *et al.*, our tool does enable the selective visualization of certain components of the solution (e.g. the solution on a boundary of the space domain, or the solution at a material interface).

# Chapter 2

# Software Architecture and Functionality

## 2.1 Architecture

Our software pipeline is divided into three sections: Pre-processing, Simulation, and Post-processing. The pre-processing and simulation sections has two modules: the Physics module and the Meshing module. During pre-processing, the Meshing module sets up the initial space mesh and the Physics module sets up the initial conditions for the PDE. During the simulation, the Meshing module generates a series of patches and passes them to the Physics module to solve. After the Physics module solves each patch, the Meshing module updates the space mesh and stores the patch solution data for later use. The post-processing section has a Pre-renderer and a Renderer, as well as a physics component. During post-processing, the pre-renderer reads in the output from the simulation, strips out all information about patches, and outputs a series of simplices to be visualized. The pre-renderer also calls the physics component to convert the physics data into a standard polynomial basis which can be understood by the renderer. The renderer visualizes the set of simplices output by the pre-renderer.

Any one of the modules can be altered independently of the others. For instance, in order to change what physics problem the simulation solves, we must only change the Physics modules. The Meshing module, Pre-renderer, and Renderer do not need to be changed because they have no information about what physics problem is being solved. Similarly, the post-processing module does not need to know what physics problem is being solved in order to render the solution data; it simply asks the physics modules for any physics-related information it needs.

## 2.2 Pre-Processing

The first stage of our SDG software is to generate a space mesh and set up the initial conditions. The space mesh can be generated using any existing tetrahedral mesh generator; we used TetGen [50, 51]. We convert the output of TetGen into our own internal mesh data format. This format consists of three files. The "vertex file", contains a list of the vertices, with each vertex assigned a unique ID and the coordinates listed in the file. The "cell file" contains a list of each of the non-vertex cells in the mesh and the IDs of their facets. The "label" file indicates for each cell whether it is on a boundary, and if so, which boundary. The Physics module takes in the vertex, cell, and label files, as well as a physics configuration file, and generates a "physics file" that contains information about the initial conditions.

Alternatively, a Python script is also provided that can generate one of several different types of test meshes. such as the *cube mesh* shown in Figure 2.1.

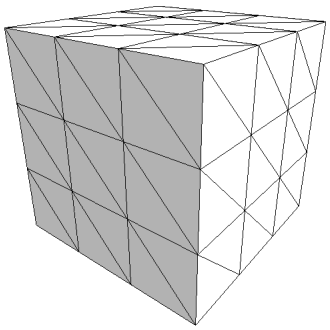

**Figure 2.1.** A $3 \times 3 \times 3$ cube test mesh.

## 2.3 Main Simulation

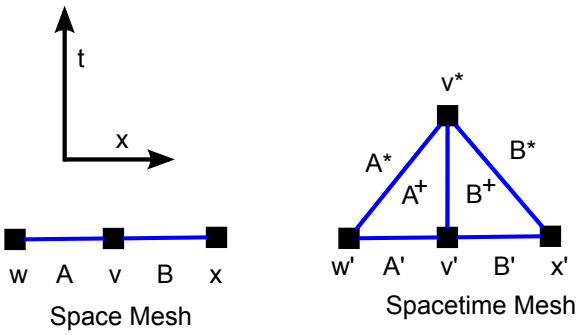


**Figure 2.2.** Example of space and spacetime cells.

The simulation code maintains a space mesh $M$ and generates a sequence of patches, each of which is part of the spacetime mesh. Each cell in the space mesh corresponds to an equivalent cell on the front of the spacetime mesh. At the beginning of the algorithm, we generate a spacetime cell corresponding to each cell in the initial space mesh; these cells form the initial front. The TentPitcher algorithm repeatedly generates patches until all of the vertices on the front are past a target time value. After each patch is solved, the front is updated by removing the inflow facets and adding the outflow facets. We update $M$ by changing the time value of the pitched vertex and the pointers that identify which spacetime cells correspond to which space cells, so that the corresponding spacetime cells of $M$ are the new front.

We also maintain a set of *predecessor* and *successor* relationships among cells. Consider any causal inflow or outflow facet $F$. If there is a cofacet $G$ of $F$ that is below $F$, then we say that $G$ is a predecessor of $F$. If there is a cofacet $H$ of $F$ that is above $F$, then we say that $F$ is a predecessor of $H$. If both of these statements are true, then we additionally say that $G$ is a predecessor of $H$. Also, $F$ is a successor of $G$ if and only if $G$ is a predecessor of $F$. These successor and predecessor relationships help the Physics code in propagating physics data, and help the Meshing code determine which inactive cells to add to the patch.

For instance, consider the situation in Figure 2.2 in $1d \times$time, with no previous patches having been

generated. Initially, cells $v, w, x, A$, and $B$ in the space mesh correspond to cells $v', w', x', A'$ and $B'$ in the spacetime mesh. Then vertex $v$ is pitched to a higher time value, which generates a new vertex $v^\star$ in the spacetime mesh. The active cells of the new patch are the newly generated outflow facets $A^\star$ and $B^\star$, as well as the newly generated extrusion cells $A^+$ and $B^+$. The new patch also contains as inactive cells the inflow facets $A'$ and $B'$. If $A'$ and $B'$ had predecessors from previous patches, those predecessors would also be inactive cells in the new patch. After the patch is solved, we update the time value on vertex $v$, and the corresponding spacetime cells of A and B become $A^\star$ and $B^\star$ respectively.

The main loop of our simulation works as follows. First, the Meshing code generates a patch by pitching a vertex with a local minimum time value. This patch is given to the Physics module to solve. After the patch is solved, the Meshing code outputs the geometry information of the new patch, and the Physics code outputs the physics solution. We then store these new results and update the front as necessary. The new results are stored in a "physics patch data" class which is opaque to the Meshing module. Each active cell in this patch contains a pointer to the physics patch data for that patch, so that when each cell is passed to future patches (as an inactive cell) the Physics module can retrieve the solution data for the cell.

We do not actually store the entire spacetime mesh; rather, cells are deleted to free up space. We say a cell is *orphaned* when it is neither on the front nor a predecessor of anything on the front. Once a cell is orphaned, it cannot be a member of any future patch. We delete any orphaned cell when all of its cofacets are deleted, so that the remaining cells always form a simplicial complex.) Once all the cells in a given patch have been deleted, that patch's solution data is also deleted.

## 2.4 Output and Visualization

The simulation produces two output streams: a *space* output stream, which shows the evolution of the spacetime mesh, and a *spacetime* output stream, which gives a sequence of patches and their corresponding solution data. The post-processing procedure consists of two components. The first is a *pre-renderer*, which takes the output streams and converts them into a series of simplices, each with one or more associated functions, which can be visualized. The second is a *visualizer*, which produces a visual representation of those simplices. The visualizer can also produce a visualization of the evolution of the space mesh by itself, with no solution data. The capabilities of the visualizer are described in more detail in section 4.1.

# Chapter 3

# Tent Pitcher Algorithm

## 3.1 Notation and Definitions

With each $d$-simplex $\Delta = \langle v_0, v_1 \dots v_d \rangle$ in $M$, we associate a *time vector* $(t_0, t_1 \dots t_d) \in \mathbb{R}^{d+1}$ listing the time coordinates of the vertices of $\Delta$; each $t_i$ is the time coordinate of the corresponding vertex $v_i$. We refer to the pair $(\Delta, \mathbf{t})$ as a *facet* of the advancing front. For any $d$-simplex $(\Delta, \mathbf{t})$, we let $\tau(\Delta, \mathbf{t}) : \mathbb{R}^d \to \mathbb{R}$ be the unique affine function that agrees with the time coordinates at the vertices of $\Delta$. Thus, $\tau(\Delta, \mathbf{t})$ agrees with the time function of the mesh over the cell $\Delta$. Thus $(\Delta, \mathbf{t})$ is *causal* if and only if $\|\nabla\tau(\Delta, \mathbf{t})\|_2 < 1$.

For any time vector $\mathbf{t}$ and any index $i$, let $t_{(i)}$ denote the $(i+1)$th smallest coordinate of $\mathbf{t}$, breaking ties arbitrarily; for example, $(3, 1, 5, 2)_{(0)} = 1$ and $(3, 1, 5, 4)_{(2)} = 4$. For any time vector $\mathbf{t}$ and any real value $x$, let $\mathbf{t} \uparrow x$ denote the vector obtained from $\mathbf{t}$ by replacing its smallest coordinate $t_{(0)}$ with $\max\{t_{(0)}, x\}$, and let $\mathbf{t} \Uparrow x$ denote the vector obtained from $\mathbf{t}$ by replacing *every* coordinate $t_i$ with $\max\{t_i, x\}$. For example, $(3, 1, 5, 4) \uparrow 4 = (3, 4, 5, 4)$ and $(3, 1, 5, 4) \Uparrow 4 = (4, 4, 5, 4)$.

## 3.2 Generic Progress Constraints

Suppose that for any $d$-simplex $\Delta$, we can define a set $\mathit{Valid}(\Delta) \subseteq \mathbb{R}^{d+1}$ of time vectors that satisfies the following conditions; to simplify notation, we say that $(\Delta, \mathbf{t})$ is *valid* if and only if $\mathbf{t} \in \mathit{Valid}(\Delta)$:

- **Openness:** The set $\mathit{Valid}(\Delta)$ is open.
- **Convexity:** The set $\mathit{Valid}(\Delta)$ is convex.
- **Causality:** If $(\Delta, \mathbf{t})$ is valid, then $(\Delta, \mathbf{t})$ is causal.
- **Progressivity:** If $(\Delta, \mathbf{t})$ is valid, then $(\Delta, \mathbf{t} \Uparrow t_i)$ is valid for all $i$.

We now formally define the TentPitcher algorithm, and show that these four abstract conditions allow the TentPitcher algorithm to mesh an arbitrary spacetime volume. The input to TentPitcher consists of the space mesh $M$ and a vector $\mathbf{S}$ listing the *starting* time coordinates of every vertex of $M$, and an arbitrary real parameter $0 < \delta < 1/2$. TentPitcher assumes that every facet of the starting front is valid.

```
TentPitcher(M, S, δ)
  T ← S
  repeat forever:
    v ← arbitrary local minimum vertex of M
    newt ← ∞
    for each facet (Δ, t) that contains v
      newt*_Δ ← sup{x | (Δ, t ↑ x) is valid}
      ρ ← arbitrary value in (δ, 1 − δ)
      newt_Δ ← ρ · newt*_Δ + (1 − ρ) · t_(1)
      newt ← min{newt, newt_Δ}
    T[v] ← newt
```

We refer to each iteration of the outer loop of TentPitcher as one *pitch*. We emphasize that for purposes of analysis, the local minimum vertex $v$ and the parameter $\rho$ can be chosen arbitrarily, or even by a malicious adversary. Larger values of $\rho$ increase progress in the current iteration, but leave less room for progress in future iterations; in practice, choosing $\rho$ close to $1/2$ seems to strike the best balance between these two concerns.

The four conditions on $\mathit{Valid}(\Delta)$ imply that for any valid facet $(\Delta, \mathbf{t})$, some open neighborhood of the line segment from $\mathbf{t}$ to $\mathbf{t} \uparrow t_{(1)}$ lies in $\mathit{Valid}(\Delta)$. It follows immediately that in each iteration of the *inner* loop of TentPitcher, we have $\mathit{newt}^*_\Delta > t_{(1)}$, and therefore $\mathit{newt}_\Delta > t_{(1)}$. Thus, every iteration of TentPitcher advances some local minimum vertex $v$ to a time value strictly larger than at least one of its neighbors in $M$, and after every iteration, every facet of the front is valid. In less formal terms, our abstract version of TentPitcher never gets stuck.

A set $\mathit{Valid}(\Delta)$ satisfying these conditions can be defined as follows. For any simplex $\Delta$, let $\mathit{Causal}(\Delta)$ denote the set of time vectors $t$ such that $(\Delta, \mathbf{t})$ is causal. Since the map from $t$ to $\nabla\tau(S, \mathbf{t})$ is linear, and the unit ball is open and convex, $\mathit{Causal}(\Delta)$ is also open and convex. We define $\mathit{Valid}(\Delta)$ to be the set of time vectors $\mathbf{t}$ such that $(\mathbf{t} \Uparrow x)$ is causal for all $x$. Since $\mathit{Valid}(\Delta)$ is the intersection of sets that are also open and convex, $\mathit{Valid}(\Delta)$ is also open and convex. By construction, $\mathit{Valid}(\Delta)$ is causal and progressive.

## 3.3 Strong Progress Guarantee

The preceding analysis leaves open the possibility that TentPitcher could execute a sequence of exponentially decreasing pitches, and therefore might never progress beyond some time value. In this section, we prove that this convergent behavior is impossible; TentPitcher moves any connected front beyond any desired time value in a finite number of iterations. Our proof is an unfortunately delicate inductive argument; readers more interested in the practical aspects of our algorithm are invited to skip ahead to Section 3.4.

For any time vector $\mathbf{t} \in \mathbb{R}^{d+1}$ and any real $\varepsilon > 0$, we define $B(\mathbf{t}, \varepsilon) = \prod_{i=0}^{d} [t_i, t_i + \varepsilon]$; in other words, $B(\mathbf{t}, \varepsilon)$ is an axis-aligned $(d+1)$-dimensional hypercube of width $\varepsilon$, whose lowest vertex in every dimension is $\mathbf{t}$. For any pair of time vectors $\mathbf{t}$ and $\mathbf{u}$, let $B(\mathbf{t}, \mathbf{u}, \varepsilon)$ denote the union of $\varepsilon$-boxes based at every point on the line segment $\overline{\mathbf{tu}}$; more formally, we have

$$B(\mathbf{t}, \mathbf{u}, \varepsilon) := \bigcup_{\lambda \in [0,1]} B(\lambda\mathbf{t} + (1-\lambda)\mathbf{u}, \varepsilon).$$

For notational convenience, let $\mathbf{t}_{(d+1)} = \infty$. Finally, for any $d$-simplex $\Delta$, any time vector $\mathbf{t}$, and any index $0 \le i \le d$, we define

$$\varepsilon_i(\Delta, \mathbf{t}) := \sup\{\varepsilon \mid B(\mathbf{t} \Uparrow t_{(i)}, \mathbf{t} \Uparrow t_{(i+1)}, \varepsilon) \in \mathit{Valid}(\Delta)\}.$$

Our assumption that $\mathit{Valid}(\Delta)$ is open and convex immediately implies that $\varepsilon_i(\Delta, \mathbf{t}) > 0$ for any valid pair $(\Delta, \mathbf{t})$ and any index $i$.

During any iteration of TentPitcher, $\Delta$ is the *binding simplex* if $\mathit{newt} = \mathit{newt}_\Delta$. We say that $\Delta$ satisfies the *binding pitch condition* if $\Delta$ is the binding simplex infinitely often when TentPitcher is run forever. The following inductive argument is the core of our proof.

**Lemma 1.** *Let $(\Delta, \mathbf{s})$ be a valid simplex that satisfies the binding pitch condition, and let $i$ be any integer such that $0 \le i \le d$. Without loss of generality, assume the vertices of $\Delta$ are indexed so that $s_{(j)} = s_j$ for all $j$. After a finite number of iterations of* TentPitcher*, we have $\min\{t_0, t_1, \ldots, t_j\} \ge s_j$ and $\max\{t_0, t_1, \ldots, t_j\} \ge s_i + \delta \cdot \varepsilon_i(\Delta, \mathbf{s})/2$ for some index $j \ge i$.*

**Proof:** We prove the lemma by induction on $i$. To simplify notation, we write $\varepsilon_i = \varepsilon_i(\Delta, \mathbf{s})$ for all $i$.

First suppose $i = 0$. Consider the first iteration of TentPitcher in which $\Delta$ is the binding simplex, and let $\mathbf{t} \in \mathit{Valid}(\Delta)$ be the time vector of $\Delta$ when that iteration begins. Let $k$ be the largest index such that $t_k > s_k$. Because each pitch increases only the minimum time coordinate of $\Delta$, we have $t_j > s_k$ for all $j \le k$ and $t_\ell = s_\ell$ for all $\ell > k$. We define two axis-aligned boxes

$$B_{in} := B(\mathbf{s}, \delta\varepsilon_0) \quad \text{and} \quad B_{out} := B(\mathbf{s}, (1-\delta)\varepsilon_0).$$

The definition of $\varepsilon_0$ implies that $B_{in} \subset B_{out} \subset \mathit{Valid}(\Delta)$. There are two cases to consider.

- Suppose $\mathbf{t} \notin B_{in}$. The definition of $B_{in}$ implies that $t_j > s_j + \delta\varepsilon_0 \ge s_0 + \delta\varepsilon_0/2$ for some index $0 \le j \le k$, even before the iteration begins.

- Suppose $\mathbf{t} \in B_{in}$. Let $j$ be the index such that $t_j = t_{(0)}$; note that $j \le k+1$. The vector $\mathbf{t}\uparrow(s_j + (1-\delta)\varepsilon_0)$ lies on the boundary of $B_{out}$ and thus in $\mathit{Valid}(\Delta)$. It follows that $\mathit{newt}^*_\Delta > s_j + (1-\delta)\varepsilon_0$, which implies that $t_j$ is changed to

$$\begin{aligned}
\mathit{newt} = \mathit{newt}_\Delta &= \rho \cdot \mathit{newt}^*_\Delta + (1-\rho) \cdot t_{(1)} \\
&> \delta \cdot \mathit{newt}^*_\Delta + (1-\delta) \cdot t_{(1)} \\
&\ge \delta \cdot \mathit{newt}^*_\Delta + (1-\delta) \cdot s_j \\
&\ge \delta \cdot (s_j + (1-\delta)\varepsilon_0) + (1-\delta) s_j \\
&= s_j + \delta(1-\delta)\varepsilon_0 \\
&\ge s_j + \delta\varepsilon_0/2. \\
&\ge s_0 + \delta\varepsilon_0/2.
\end{aligned}$$

  The second-to-last inequality relies on our assumption that $\delta < 1/2$.

In both cases, when the iteration ends, we have $\min\{t_0, t_1, \ldots, t_j\} \ge s_j$ and $\max\{t_0, t_1, \ldots, t_j\} > s_0 + \delta\varepsilon_0/2$, for some index $j \ge 0$, as required. This completes the proof when $i = 0$.

Now suppose $i > 0$. For purposes of analysis, we divide the execution of the algorithm into three stages. Stage 1 ends when $\max\{t_0, \ldots, t_{i-1}\} \geq s_i$. Stage 2 ends when $\min\{t_0, \ldots, t_i\} \geq s_i$. Finally, Stage 3 ends when the concluding conditions of the lemma are satisfied: $\min\{t_0, \ldots, t_j\} > s_j$ and $\max\{t_0, \ldots, t_j\} > s_i + \delta\varepsilon_i/2$ for some index $j \geq i$. We argue separately that each stage ends after a finite number of iterations.

**Stage 1:** Let $t^* = \max\{t_0, \ldots, t_{i-1}\}$. The value $t^*$ increases over time during the execution of TentPitcher. We have $t^* = s_{i-1}$ initially, and Stage 1 ends when $t^* \geq s_i$. For purposes of analysis, we further divide Stage 1 into *phases*, where each phase either ends Stage 1 or increases $t^*$ by at least $\delta\varepsilon_{i-1}/2$. Stage 1 ends after at most $\lceil 2(s_i - s_{i-1})/\delta\varepsilon_{i-1} \rceil$ phases; thus, it suffices to prove that each phase requires only a finite number of pitches.

Consider an iteration of TentPitcher in which $\Delta$ is the binding simplex, and let $\mathbf{t}$ be the time vector of $\Delta$ when that iteration begins. If $t^* \geq s_i$, Stage 1 is already over. Otherwise, $t_0, t_1, \ldots, t_{i-1}$ are still the lowest $i$ coordinates of $\mathbf{t}$; in particular, $t^* = t_{(i-1)} \geq s_{i-1}$. It follows that $\mathbf{t} \Uparrow x = \mathbf{s} \Uparrow x$ for all $x \geq t_{(i-1)}$, which in turn implies that $\varepsilon_{i-1}(\Delta, \mathbf{t}) \geq \varepsilon_{i-1}(\Delta, \mathbf{s})$. The inductive hypothesis implies that after a finite number of pitches, we have $\min\{t_0, t_1, \ldots, t_j\} \geq s_j$ and $\max\{t_0, t_1, \ldots, t_j\} \geq s_{i-1} + (\delta/2) \cdot \varepsilon_{i-1}(\Delta, \mathbf{t}) \geq s_{i-1} + (\delta/2) \cdot \varepsilon_{i-1}(\Delta, \mathbf{s})$ for some index $j \geq i-1$. If $j \geq i$, then $t^* \geq \min\{t_0, t_1, \ldots, t_j\} \geq s_j \geq s_i$, and Stage 1 is complete. Otherwise, if $j = i - 1$, then $t^* = \max\{t_0, t_1, \ldots, t_j\}$ has increased by at least $\delta\varepsilon_{i-1}/2$, ending the phase.

**Stage 2:** Let $\mathbf{u}$ denote the time vector of $\Delta$ just after Stage 1 ends. At most $i - 1$ elements of $\mathbf{u}$ are smaller than $s_i$, which implies that $u_{(i-1)} \geq s_i$. Thus, the inductive hypothesis implies that after a finite number of pitches, we have $\min\{t_0, \ldots, t_i\} \geq u_{(i-1)} \geq s_i$, at which point Stage 2 ends.

**Stage 3:** The proof for Stage 3 closely follows the argument for the case $i = 0$. Consider the first iteration of TentPitcher after Stage 2 ends in which $\Delta$ is the binding simplex, and let $\mathbf{t} \in \mathit{Valid}(\Delta)$ be the time vector of $\Delta$ when that iteration begins. Let $k$ be the largest index such that $t_k > s_k$. Because Stage 2 has ended, we must have $k \geq i$ and $\min\{t_0, t_1, \ldots, t_k\} > s_k$. Because each pitch increases only the minimum time coordinate of $\Delta$, we also $t_\ell = s_\ell$ for all $\ell > k$. We define two axis-aligned boxes $B_{in} = B(\mathbf{s} \Uparrow s_i, \delta\varepsilon_i)$ and $B_{out} = B(\mathbf{s} \Uparrow s_i, (1-\delta)\varepsilon_i)$. The definition of $\varepsilon_i$ implies that $B_{in} \subset B_{out} \subset \mathit{Valid}(\Delta)$.

If $\mathbf{t} \notin B_{in}$, then $t_j > s_j + \delta\varepsilon_i \geq s_i + \delta\varepsilon_i/2$ for some index $0 \leq j \leq k$, and so Stage 3 has already ended.

On the other hand, suppose $\mathbf{t} \in B_{in}$. Let $\ell$ be the index such that $t_\ell = t_{(0)}$, and let $j = \max\{i, \ell\}$. We have both $j \leq k + 1$ and $t_{(0)} \geq s_j$. The vector $\mathbf{t} \uparrow (s_j + (1-\delta)\varepsilon_i)$ lies on the boundary of $B_{out}$ and thus in $\mathit{Valid}(\Delta)$. It follows that $\mathit{newt}^*_\Delta > s_j + (1-\delta)\varepsilon_i$, which implies (following the same logic as $i = 0$) that $\mathit{newt} = \mathit{newt}_\Delta \geq s_j + \delta\varepsilon_i/2 \geq s_i + \delta\varepsilon_i/2$. Thus, when this iteration ends, we have $\min\{t_0, t_1, \ldots, t_j\} \geq s_j$ and $\max\{t_0, t_1, \ldots, t_j\} > s_i + \delta\varepsilon_i/2$, and therefore Stage 3 has ended. □

**Theorem 1.** *Let $M$ be an arbitrary connected mesh, and let $t_{\max}$ be any real number.* TentPitcher *pitches all time values in $M$ above $t_{max}$ after a finite number of iterations.*

**Proof:** For any two vertices $v$ and $w$ in $M$, let $d(v, w)$ denote the length of the shortest path from $v$ to $w$ in the 1-skeleton of $M$. Let $D = \max_{v,w} d(v, w)$ denote the diameter of the 1-skeleton of the input mesh $M$. Because every simplex in $M$ is causal, the time values of any two vertices $v$ and $w$ differ by at most $d(v, w) \leq D$. Thus, it suffices to prove that after a finite number of pitches, at least one vertex of $M$ has time value larger than $t_{max} + D$. The proof closely follows the argument for Stage 1 in the proof of Lemma 1.

Let $\Delta$ be any $d$-simplex in $M$ that satisfies the binding pitch condition; at least one such simplex exists. Let $\mathbf{s}$ be the starting time vector of $\Delta$, and assume without loss of generality that the vertices of $\Delta$ are indexed so that $s_{(i)} = s_i$. Let $\mathbf{t}$ denote the time vector of $\Delta$ during the algorithms execution, and let $t^* = \max\{t_0, t_1, \ldots, t_d\}$; initially, we have $t^* = s_d$.

For purposes of analysis, we divide the execution of the algorithm into *phases*, where each phase increases $t^*$ by at least $\delta\varepsilon_d(\Delta, \mathbf{s})/2$. Because $\mathbf{t} \Uparrow x = \mathbf{s} \Uparrow x$ for all $x \geq t^*$, we have $\varepsilon_d(\Delta, \mathbf{t}) \geq \varepsilon_d(\Delta, \mathbf{s})$. Thus, Lemma 1 (with $i = d$) immediately implies that each phase ends after a finite number of pitches. After at most $\lceil 2(t_{max} + D - s_d)/\delta\varepsilon_d(\Delta, \mathbf{s}) \rceil$ phases, we have $t^* > t_{max} + D$, as required. □

## 3.4 Refinement

We now discuss how to modify TentPitcher to support refinement. We first review refinement in 2 dimensions. Next, we describe Bänsch refinement, a generalization of newest-vertex bisection that applies to 3 dimensions. We derive a set of progress constraints that will enable Bänsch refinement to work. Finally, we show how to integrate Bänsch refinement into TentPitcher and show that the adaptive version of TentPitcher advances the front arbitrarily far forward in time.

### 3.4.1 2D Refinement

In 2 dimensions, following Abedi *et al.* [2], we use the *newest-vertex bisection* method of Sewell [47] and Mitchell [36, 37, 38]. Each triangle in the mesh maintains a *marked edge*. Triangles are bisected by a segment from the midpoint of the marked edge to the opposite vertex; in each of the resulting triangles, the edge furthest from the midpoint of the previous marked edge is marked. We refer to the simplices generated by bisecting a simplex as the *children* of that simplex.

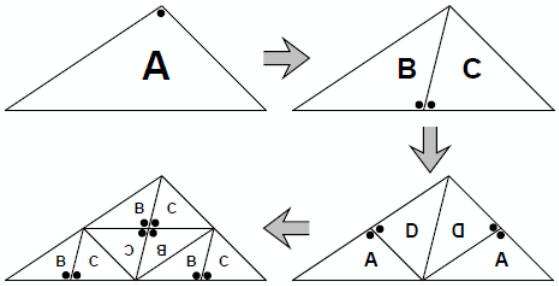


**Figure 3.1.** Newest-vertex refinement of a single triangle. Letters indicate which homothety class each child belongs to. Figure taken from Abedi *et al.* [2].

Suppose that a triangle $abc$ with marked edge $bc$ is bisected. If $bc$ is not on the boundary, then there exists some neighboring triangle $cbe$ that shares that edge, so in order to maintain a conforming mesh $cbe$ must be bisected as well. If $bc$ is also the marked edge of $cbe$, then the resulting mesh will be a simplicial complex. However, if the marked edge of $cbe$ is not $bc$, and that marked edge is not on the boundary, then the other triangle that includes that marked edge must also be bisected. In this way, the refinement propagates recursively. Refinement continues until the mesh is a simplicial complex, which always occurs after a finite

number of iterations and usually takes only a small number of iterations [2]. The children of each simplex all belong to one of eight homothety classes, which can be used to derive the progress constraints [2].

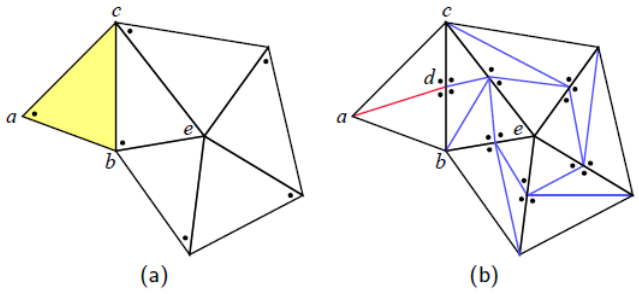


**Figure 3.2.** Newest-vertex refinement propagating to neighboring tetrahedra. Once *abc* is bisected, *cbe* must be bisected as well, and the bisection propagates recursively. Figure taken from Abedi *et al.* [2]

As we will see below, adaptivity requires that in addition to the constraints of openness, convexity, causality, and progressivity, we must also satisfy the *inheritance* condition: if any simplex is valid then its children are also valid. Abedi *et al.* [2] derive a set of progress constraints based on these homothety classes that satisfy the conditions described in Section 3.2. We fix a real number $0 < \omega < 1$, and define the *diminished width* $w(abd)$ of a triangle $abd$ to equal $1 - \omega$ times the minimum of all the altitudes of the triangle. The progress constraints are as follows:

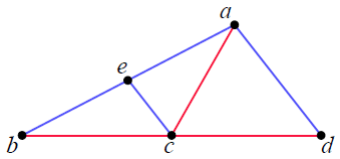


**Figure 3.3.** Subdivisions of cell for 2d refinement progress constraints. Figure taken from Abedi *et al.* [2].

Given a triangle $abd$ where $bd$ is the marked edge, we let $c$ be the midpoint of $bd$ and let $e$ be the midpoint of $ab$. Let $t(a)$ be the time value at vertex $a$, let $t(b)$ be the time value at vertex $b$, and so on. Then a simplex is valid if and only if:

1. The simplex is causal.
2. $|t(a) - t(b)| < 2w(cad)$
3. $|t(a) - t(c)| < w(abd)$
4. $|t(a) - t(d)| < 2w(cab)$
5. $|t(b) - t(d)| < 4w(eca)$

The topology, geometry, and causality conditions are trivially true by construction. Abedi *et al.* [2] show that if the lowest vertex is raised to the time value of the second-lowest vertex, the simplex will remain valid. This condition is sufficient to ensure progressivity.

### 3.4.2 Bänsch Refinement

To generate adaptive meshes in 3d×time, our new algorithm uses a three-dimensional generalization of newest-vertex bisection first described by Bänsch [11, 12], and independently developed and/or extended by Arnold *et al.* [10], Kossaczk [26], Liu and Joe [28, 29], Maubach [31, 32], Moore [39], Stephenson [52], and Traxler [57]. We refer to this technique as *Bänsch refinement*, although our description and notation more closely follows later authors [10, 28, 29].

As in newest-vertex bisection, we associate a marked edge with every *triangle* in the mesh, with the restriction that at least two of the facets of any tetrahedron $\Delta$ must mark the same edge of $\Delta$. We refer to the edge of $\Delta$ that is marked twice as the *refinement edge* of $\Delta$; if there are two such edges, we choose one arbitrarily. There are four valid marking patterns, illustrated in Figure 3.4; we adopt the notation of Arnold *et al.* [10] to describe these patterns:

1. **Planar ($P$):** The marked edges are coplanar.
2. **Adjacent ($A$):** The marked edges define a path of length 3, with the refinement edge in the middle.
3. **Mixed ($M$):** The marked edges define a path of length 3, with the refinement edge at one end.
4. **Opposite ($O$):** The edge opposite the refinement edge is also marked twice.

Tetrahedra of type $P$ also have a boolean flag associated with them which can be either unset ($P_u$) or set ($P_f$), giving a total of five different marking types. We refer to tetrahedra of type $P_f$ as *flagged*.

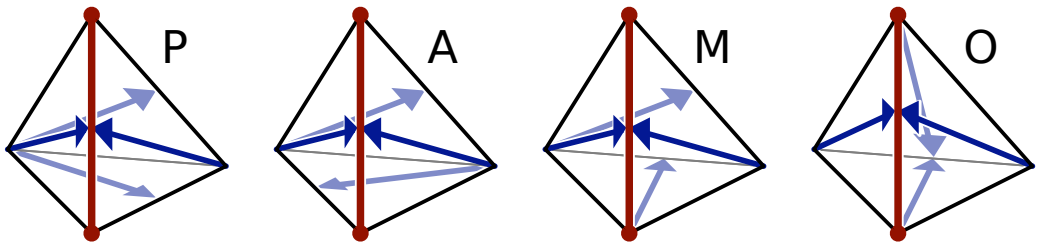


**Figure 3.4.** The four valid marking patterns for tetrahedra. In each figure, arrows indicate the marked edge for each facet, and the bold vertical segment is the refinement edge.

To refine a tetrahedron $\Delta$, we bisect it along the plane through the midpoint of the refinement edge and two vertices opposite the refinement edge. Two of the facets of $\Delta$ are bisected; each of the four resulting subfacets mark the edge furthest from the midpoint of the refinement edge. The marked edge for the triangle in the interior of $\Delta$ is chosen so that the marking types of the children of $\Delta$ depends on the marking types of $\Delta$, as follows:

- If $\Delta$ has type $P_u$, both its children have type $P_f$.
- If $\Delta$ has type $P_f$, both its children have type $A$.
- If $\Delta$ has type $A$, $M$, or $O$, both its children have type $P_u$.

This marking schedule guarantees that descendants of each simplex fall into a finite number of homothety classes [10].

If the bisected simplex $\Delta$ is part of a larger mesh, any other tetrahedron that contains the refinement edge must also be bisected. This bisection propagates through the mesh in a manner similar to newest-vertex bisection refinement. Arnold *et al.* [10] prove the following theorem:

**Theorem 2.** *Consider a mesh $S_0$ that contains no flagged tetrahedra. Consider a sequence of meshes $S_1 \dots S_j$ such that each $S_i$ is obtained by refining a single tetrahedron in $S_{i-1}$ (in addition to any further bisections due to the propagation.) Then when we refine a single tetrahedron in $S_j$, the refinement will always terminate.*

Although it is theoretically possible that the bisection will propagate through the entire mesh, in practice the number of bisections is usually a small constant.

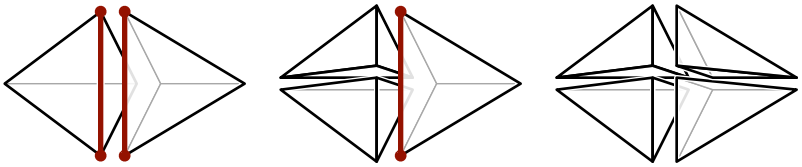

**Figure 3.5.** Bänsch refinement propagating to a neighboring tetrahedron. Refinement edges are bold.

There are several methods to compute a valid marking of the triangles in an arbitrary tetrahedral mesh $M$ that leaves all type $P$ tets unflagged. Our implementation simply marks the longest edge in each triangle of $M$; this strategy guarantees that the longest edge of any tetrahedron is the refinement edge for that tetrahedron.

## 3.5 Adaptive Progress Constraints

We now derive adaptive progress constraints for the adaptive case that satisfy openness, convexity, causality, and progressivity, and inheritance. We refer to these progress constraints as the *adaptive progress constraints*, and we say that a simplex that satisfies these progress constraints is *adaptively valid*. We first define adaptive progress constraints for a set of *standard reference tetrahedra*. Then, we show how to generalize these progress constraints to arbitrary tetrahedra.

### 3.5.1 Reference Tetrahedra

First, we define a *standard reference tetrahedron* for each of the three mark types $A, P_u$, and $P_f$. (We discuss mark types $O$ and $M$ in the next section.) These reference tetrahedra are based on the mapping used by Liu and Joe [28, 29]. For each of the annotations $A, P_u, P_f$, we define standard reference tetrahedra $\mathbf{A}, \mathbf{P_u}$, and $\mathbf{P_f}$ as follows.

$$\mathbf{A} \equiv \langle(-1,-1,-1),(1,-1,-1),(1,1,-1),(1,1,1)\rangle$$
$$\mathbf{P_u} \equiv \langle(-1,-1,-1),(1,-1,-1),(1,1,-1),(0,0,0)\rangle$$
$$\mathbf{P_f} \equiv \langle(-1,-1,-1),(1,-1,-1),(0,0,-1),(0,0,0)\rangle$$

These tetrahedra are marked as shown in Figure 3.6; in each facet, the longest edge is marked.

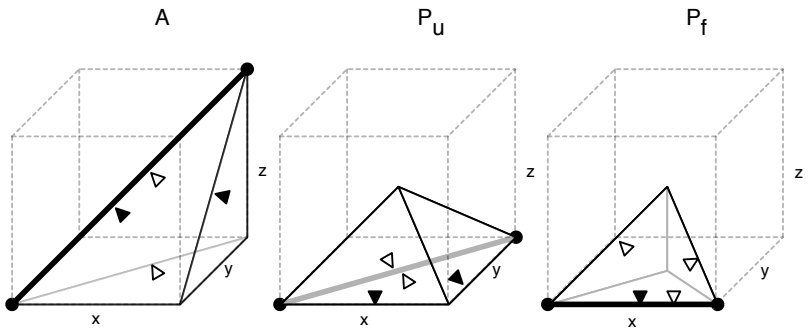


**Figure 3.6.** Standard reference tetrahedra. $\mathbf{A}, \mathbf{P_u}$, and $\mathbf{P_f}$, correspond to the simple cubic, body-centered cubic, and face-centered cubic lattice structures respectively. Triangles indicate the marked edge of each face; black triangles indicate marked edges of faces closer to the viewer while white triangles indicate marked edges of faces farthest from the viewer.

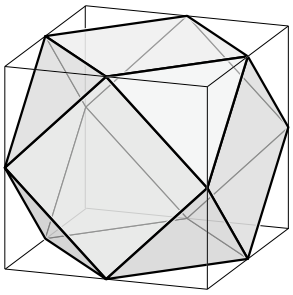

**Figure 3.7.** 3d×time progress constraints.

We say that a reference tetrahedron $(R, \mathbf{t})$ is *adaptively valid* if and only if $\|\nabla\tau(R, \mathbf{t})\|_1 < \sqrt{2}/2$ and $\|\nabla\tau(R, \mathbf{t})\|_\infty < \sqrt{2}$ (Figure 3.7). These conditions define a cuboctahedron $C$ in $\mathbb{R}^3$ which must contain $\tau(S, \mathbf{t})$.

If a tetrahedron has two vertices with the same time value, $\nabla\tau(R, \mathbf{t})$ is normal to the edge joining those two vertices. All of the edges of the reference tetrahedra are parallel to one of the vectors that can be obtained from $(1, 0, 0), (1, 1, 0)$, or $(1, 1, 1)$ by a sequence of permutations or reflections about coordinate axes. The 13 planes normal to these vectors partition space into 96 *elementary cones* (Figure 3.8).

Consider any reference tetrahedron $(R, \mathbf{t})$. The vertex of $R$ with the lowest time value (the *lowest vertex* of $R$) can be determined solely from $\nabla\tau(R, \mathbf{t})$. Thus, we partition the space of possible values of $\nabla\tau(R, \mathbf{t})$ into four regions, each of which contain the values of $\nabla\tau(R, \mathbf{t})$ that make one of the vertices lowest. Because the map from $\mathbf{t}$ to $\nabla\tau(R, \mathbf{t})$ is linear, these regions are convex, and therefore connected. Figure 3.8 shows *gradient diagrams* for each of the three reference tetrahedra. In each diagram, the space of possible gradients is shown as a cube divided into elementary cones that are shaded to indicate which region they belong to.

For a reference tetrahedron $(R, \mathbf{t})$ with a unique lowest vertex, we define the *one-vertex pitching direction* to be the direction that $\nabla\tau(R, \mathbf{t})$ moves when the lowest vertex is pitched; this direction is normal to the face opposite the lowest vertex, pointing in the direction of the lowest vertex. In Figure 3.8, the one-vertex pitching direction is represented by a large dot of the same shade as the vertex; the one-vertex pitching direction is the vector from the origin to that dot.

For a reference tetrahedron $(R, \mathbf{t})$ with two vertices tied for lowest, we define the *two-vertex pitching direction* to be the direction that $\nabla\tau(S, \mathbf{t})$ moves when both of the lowest vertices are pitched at the same

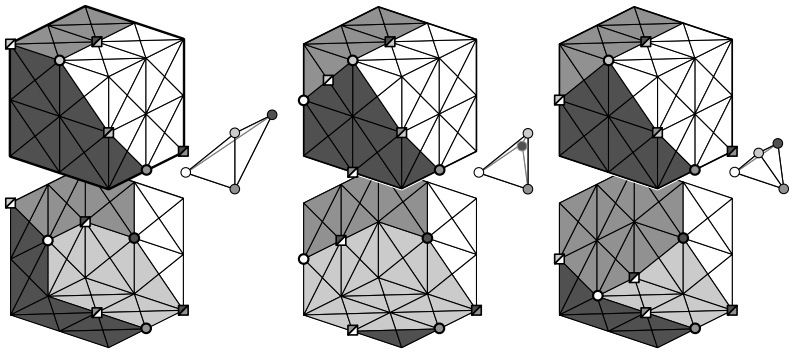

**Figure 3.8.** Gradient diagrams for three different reference tetrahedra. Large dots indicate pitching directions. Squares with slashes indicate two-vertex pitching directions.

time by the same amount. That is, the two-vertex pitching direction is the sum of the pitching directions for each of the two vertices. In Figure 3.8, the two-vertex pitching direction for each pair of vertices is represented by a box with the shade of both of those vertices.

**Lemma 2.** *Consider any reference tetrahedron $(R, \mathbf{t})$ with a unique lowest vertex. Let $E$ be the elementary cone that contains $\nabla\tau(S, \mathbf{t})$. Let $\mathbf{p}$ be the one-vertex pitching direction for $(R, \mathbf{t})$ and let $\mathbf{n}$ be the outward-pointing normal to the face of $C$ that intersects $E$. Then $\mathbf{p} \cdot \mathbf{n} \leq 0$.*

**Proof:** All values of $\nabla\tau(R, \mathbf{t})$ within $E$ have the same one-vertex pitching direction $\mathbf{p}$. Thus, this theorem can be proven by an exhaustive case enumeration, with one case for each combination of elementary cone and reference tetrahedron. □

A similar exhaustive case analysis implies the following lemma:

**Lemma 3.** *Consider any reference tetrahedron $(R, \mathbf{t})$ with two vertices tied for lowest. Let $E$ and $E'$ be two elementary cones containing $\nabla\tau(S, \mathbf{t})$. Let $\mathbf{p}$ be the two-vertex pitching direction for $(R, \mathbf{t})$. Let $\mathbf{n}$ and $\mathbf{n}'$ be the outward-pointing normals to the faces of $C$ that intersect $E$ and $E'$. Then $\mathbf{p} \cdot \mathbf{n} \leq 0$ and $\mathbf{p} \cdot \mathbf{n}' \leq 0$.*

**Lemma 4.** *For any adaptively valid reference tetrahedron $(R, \mathbf{t})$, we have that $(R, t \Uparrow t_{(1)})$ and $(R, t \Uparrow t_{(2)})$ are adaptively valid.*

**Proof:** Let $F$ be a face of $C$ intersected by the ray starting at the origin and going in the direction of $\nabla\tau(R, \mathbf{t})$. Lemma 2 implies that we cannot move $\nabla\tau(R, \mathbf{t})$ out of $C$ by pitching the lowest vertex up to $t_{(1)}$. Thus, $(R, \mathbf{t} \Uparrow t_{(1)})$ is adaptively valid.

Let $F'$ be a face of $C$ intersected by the ray starting at the origin and going in the direction of $\nabla\tau(R, \mathbf{t} \Uparrow t_{(1)})$. Lemma 3 implies that we cannot move $\nabla\tau(R, \mathbf{t})$ out of $C$ by pitching the two lowest vertices of $\tau(R, \mathbf{t} \Uparrow t_{(1)})$ up to $t_{(2)}$. Thus, $(R, \mathbf{t} \Uparrow t_{(1)})$ is adaptively valid. □

### 3.5.2 Arbitrary Tetrahedra

We now extend our adaptive progress constraints to arbitrary tetrahedra. Consider an arbitrary tetrahedron $T$ with annotation type $P_u, P_f$, or $A$, and its standard reference tetrahedron $R$ of the same annotation type. (Tetrahedra with annotation types $M$ and $O$ will be discussed later.) We order the vertices in $T$ and $R$ so the

ordering respects the edge markings. (There may be multiple valid vertex orderings; choose one arbitrarily.) We show that there is a linear map that depends only on $T$ and $R$ that maps $\nabla\tau(R, \mathbf{t})$ to $\nabla\tau(T, \mathbf{t})$.

Without loss of generality, assume that the *first* vertices of $R$ and $T$ have coordinates $(0, 0, 0)$ and time value 0. Let $M$ be the matrix of the linear transformation that maps $T$ to $R$. Then any point $p$ in $(T, \tau)$ has the same time value as the point $Mp$ in $(R, \tau)$. Thus, we we have $(\nabla\tau(T, \mathbf{t}))^\top p = (\nabla\tau(R, \mathbf{t}))^\top Mp$, which implies that $\nabla\tau(T, \mathbf{t}) = M^\top \nabla\tau(R, \mathbf{t})$.

Let $M^\top(C)$ be the image of $C$ under the transformation $M^\top$. The progress constraint for $R$, which is $\nabla\tau(R, \mathbf{t}) \in C$, is equivalent to $\nabla\tau(T, \mathbf{t}) \in M^\top(C)$. We define a *scaling factor* $q = 1/\max_{\mathbf{u}\in C}\|M^\top \mathbf{u}\|_2$,, so that $q \cdot M^\top C$ just fits inside the unit ball. Since $C$ is the convex hull of 12 points, $q$ can be calculated by checking each of those points.

Finally, we define a tetrahedron $(T, \mathbf{t})$ to be *adaptively valid* if and only if $\|\nabla\tau(R, \mathbf{t})\|_1 < q\sqrt{2}/2$ and $\|\nabla\tau(R, \mathbf{t})\|_\infty < q\sqrt{2}$. In other words, the tetrahedron is adaptively valid if and only if $\nabla\tau(R, \mathbf{t}) \in qC$.

We show that these adaptive progress constraints satisfy the openness, convexity, causality, progressivity, and inheritance conditions. Openness and convexity are trivial. If $(T, \mathbf{t})$ is adaptively valid, then $\tau(R, \mathbf{t})$ is in $qC$, so $\|M^\top \nabla\tau(R, \mathbf{t})\| < 1$ and thus $T$ is causal. Since the adaptive progress constraints for $T$ are just the progress constraints for $R$ scaled by a constant, the set of adaptively valid tetrahedra is progressive.

**Lemma 5.** *If $(T, \mathbf{t})$ is adaptively valid, then its children are adaptively valid.*

**Proof:** Let $R$ be the standard reference tetrahedron for $T$. Consider a child $T^\star$ of $T$ and the corresponding child $R'$ of $R$. Let $R^\star$ be the standard reference tetrahedron for $T^\star$, and let $\mathbf{t}^\star$ be the time vector for $T^\star$. Then $\nabla\tau(R', \mathbf{t}^\star) = \nabla\tau(R, \mathbf{t})$. Define $M_{adj}$ to equal the mapping that takes $R^\star$ to $R'$ (ignoring translations). $\nabla\tau(R^\star, \mathbf{t}^\star) = M_{adj}^\top \nabla\tau(R', \mathbf{t}^\star)$. Thus, $M_{adj}^\top$ is the product of scalings, 90-degree rotations about coordinate axes, and flips about a coordinate axes. Scaling by a factor of $x$ changes the gradient and $q$ both by a factor of $x$, so validity is unchanged. Since $C$ is symmetrical with respect to flips and 90-degree rotations about the coordinate axis, those operations also do not affect validity. We conclude that $(T^\star, \mathbf{t}^\star)$ is adaptively valid. □

We now consider cells of type $M$ or $O$. We define a tetrahedron $(S, \mathbf{t})$ with one of these types to be adaptively valid if each of its children are adaptively valid, each of the children of $(S, \mathbf{t} \Uparrow t_{(1)})$ are adaptively valid, and each of the children of $(S, \mathbf{t} \Uparrow t_{(2)})$ are adaptively valid. By construction, this definition satisfies the openness, convexity, causality, progressivity, and inheritance properties.

### 3.5.3 Algorithm

The pitching algorithm for adaptivity consists of a series of iterations. In each iteration, we choose to either pitch an arbitrary local minimum vertex or refine an arbitrary tetrahedron. As before, the decision of whether to pitch or refine, and which vertex to pitch or tetrahedron to refine, can be made arbitrarily or even by a malicious adversary. If we choose to pitch a vertex, we pitch it using one iteration of the inner loop of TentPitcher. If we choose to refine a tetrahedron, we do so using the algorithm described in section Section 3.4.2.

**Theorem 3.** *Suppose that* TentPitcherAdaptive *calls* Refine *finitely many times. Then, given any time value $t_{max}$,* TentPitcherAdaptive *pitches every vertex above $t_{max}$ in a finite number of steps.*

**Proof:** Since only a finite number of refinement operations are performed, there must be a last refinement operation. Since both refinement and pitching operations preserve progressivity, after this operation all the simplices in $M^\star$ are still valid. After that, only pitching operations are performed, so the algorithm is equivalent to TentPitcher with different progress constraints that still satisfy the conditions needed for Theorem 1: openness, convexity, causality, and progressivity. □

The constraint that after a certain point only pitching operations are performed is essential. If we were to alternate refinement and pitching operations indefinitely, then subsequent pitching operations would pitch shorter and shorter distances (since the cells get finer and finer) and could converge to a finite value.

## 3.6 Adaptivity Operations

### 3.6.1 Overview

Our infrastructure also supports other standard mesh adaptivity operations, including edge contraction, edge flipping, and smoothing. We do not currently understand how to combine these operations with refinement while maintaining the conditions of Theorem 2, although we have a method which appears to work in practice.

To determine when to perform adaptivity operations, we use a *quality metric* on each tetrahedron. The quality of a tetrahedron $T$ is defined as $det(M^\top)/q^3$. In other words, the quality of $T$ is equal to the ratio of the volume of the transformed and scaled version of $C$ that defines the progress constraints for $T$ to the volume of the original $C$. Intuitively, lower quality tets allow less progress in each pitch. The best possible quality is 1, and the worst possible quality (which can only occur with a degenerate tetrahedron that cannot be pitched at all) is 0.

Each adaptivity operation alters the space mesh by adding some simplices and removing other simplices. To perform an adaptivity operation, we generate a spacetime patch whose inflow facets are the simplices removed from the front, and whose outflow facets are the simplices added to the front. The spacetime volume in between the inflow and outflow facets is divided into tetrahedra; these are the active cells of the patch. This coarsening method was originally described by Abedi *et al.* [7, 44].

**Edge Contraction.** Edge contraction [15] is performed by contracting an edge down to a single vertex, thus removing one of the vertices. We refer to the vertex removed as the *removal vertex* $r$, and the other endpoint of the edge as the *contraction target* $s$. This operation removes each cell in the front that contained both $r$ and $s$, and it replaces each cell that contained $r$ with a cell that contains $s$. In order to perform this edge contraction, we create a patch where, for each cell $C$ containing $r$, we create a corresponding extrusion cell that has all the vertices of $C$ and also has vertex $s$, and a corresponding outflow facet that has all the vertices of $C$ except with $r$ replaced with $s$. Edge contraction is only performed when all the cells adjacent to the contraction target are marked as coarsenable.

**Edge Flipping.** We implemented three types of edge flips as described by Klingner and Shewchuk [24]:

1. 2-3 edge flipping, which removes a triangular face that separates two tetrahedron, adds an edge between the endpoints of each tetrahedron opposite the removed face, and for each vertex in the removed

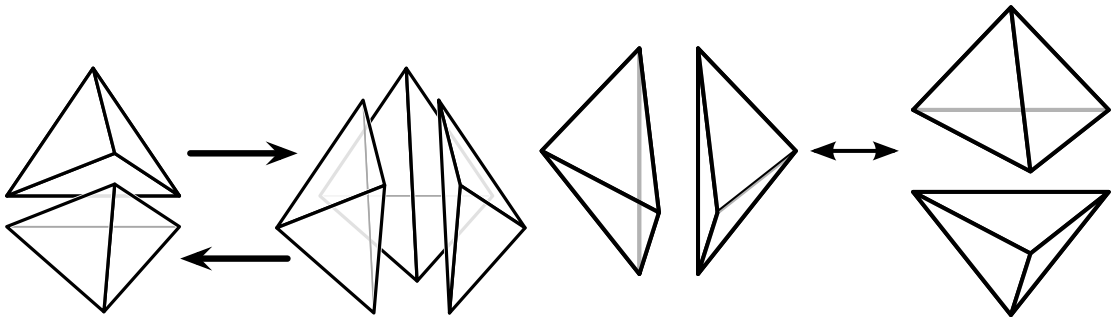

**Figure 3.9.** The three different types of edge flips. The figure on the left shows 2-3 and 3-2 edge flips, while the figure on the right shows 2-2 edge flips.

triangle, adds a triangular face that contains that vertex and the new edge.

2. 3-2 edge flipping, which removes an edge that is not on the boundary and has exactly three cofacets, removes the cofacets of that edge, and creates a triangular face whose vertices are the vertices opposite the removed edge in each of the cofacets of the removed edge.

3. 2-2 edge flipping, which removes an edge that is on the boundary and has exactly three cofacets, two of which are on the boundary. 2-2 edge flipping is similar to the standard $2d$ edge flip on the boundary of the mesh. In order to ensure that 2-2 edge flipping does not change the region that the mesh covers, it is necessary to ensure that the two cofacets on the boundary are coplanar.

Each of these edge flips can be performed by creating a patch with only one active spacetime 4-cell: the simplex whose vertices are all 5 of the vertices involved in the edge flip.

**Smoothing.** Smoothing simply changes the spatial coordinates of a given vertex in the space mesh. Patch generation for smoothing is performed in exactly the same way as patch generation for tent pitching, with the exception that the spatial coordinates are changed rather than just the time coordinates. Smoothing does not change the topology of the mesh. To ensure that smoothing does not change the region that the mesh covers, if a vertex is on the boundary it is moved only in directions coplanar to the triangles on the boundary that contain it (in the case of a 3-dimensional space mesh) or collinear to the edges on the boundary that contain it (in the case of a 2-dimensional space mesh). To smooth a vertex, we first determine what spatial coordinates to pitch it to, then determine the time value to move it to in the same way as determining the time value to pitch it to during a tent pitching operation, with the exception that the determination of whether it is okay to pitch to a given time value is done based on the new spatial coordinates instead of the old spatial coordinates.

There are several methods of determining where to move a vertex to when smoothing. The simplest method is known as Laplacian smoothing [22, 18, 19], where a vertex is moved to the centroid of its neighbors. Another method, which is more computationally expensive but can produce better results, is to search for the quality-maximizing point using a hill-climbing method [9]. (Getting to the optimal point in one smoothing operation is not essential, because there will be opportunities for further smoothing later.)

### 3.6.2 Remarking Edges for Refinement

To combine coarsening and edge flipping with refinement, we must ensure that the mesh has a legal marking after each operation. We stress that we *do not know* whether our remarking strategy always enables further refinement. Suppose we start with a mesh with no flagged tetrahedra, then refine it (generating a flagged tetrahedron), and then perform an operation that changes the markings elsewhere. The resulting mesh does not satisfy the conditions of Theorem 2. Experiments suggest that if we do not generate any new flagged tetrahedra when remarking, the refinement process always terminates.

When a triangle is remarked, we must recompute the annotations of all tetrahedra that contain it. If a tetrahedron has mark type $P$ after remarking, we assign it annotation $P_u$. Some adaptivity operations require remarking facets of tetrahedra that are not in the patch, but are nearby. We require all remarked tetrahedra to be adaptively valid.

We propose a remarking strategy called *edge collection*. To perform edge collection, we choose a sequence of edges $E$ whose union consists of disjoint paths and cycles. For each triangle that contains one or more edges in $E$, we mark the edge in $E$ that is first in the sequence.

**Lemma 6.** *Edge collection always produces a valid set of markings.*

**Proof:** Consider any tetrahedron $ABCD$. Let $x$ be the number of edges of $ABCD$ that are in $E$.

1. If $x = 0$, no remarking occurred.
2. If $x = 1$, then without loss of generality we have that $AB$ is in $E$. By construction, $ABC$ and $ABD$ both mark $AB$, so the tetrahedron is valid.
3. If $x = 2$, either the two edges in $E$ share a vertex or they do not. If they do not share a vertex, then without loss of generality assume one of the edges is $AB$. Thus, $ABC$ and $ABD$ both mark $AB$, so the tetrahedron is valid. If the edges in $E$ do share a vertex, assume without loss of generality that the edges are $AB$ and $AC$. Then $ABD$ marks $AB$, $ACD$ marks $AC$, and $ABC$ marks either $AB$ or $AC$. Either choice makes the tetrahedron valid.
4. If $x = 3$, then the three edges in $E$ cannot share a vertex. Thus, every facet of $ABCD$ contains an edge in $E$. There are only three such edges, so one must be marked by two facets.
5. If $x = 4$, then the four edges must form a cycle $(AB, BC, CD, DA)$. The edge that is first in the total order is marked by both facets that contain it.
6. We cannot have $x > 4$ because then some vertex would be incident to at least three edges in $E$.

□

To remark after edge contraction, we perform an edge collection on the *link* of the removed edge. The link of an edge $E$ is the set of all faces of cofaces of $E$ that are disjoint from E [33]. To remark after a 2-3 flip or 2-2 flip, we perform edge collection on the edge that was added. To remark after a 3-2 flip, we perform edge collection on the edges of the new triangle. No remarking is necessary after smoothing.

## 3.7 Adaptivity Workflow

We now return to our description of the software architecture, and discuss how to incorporate adaptivity. Several changes to the Physics code are necessary. The Physics code computes an estimate of the numerical error on each facet of the spacetime mesh when a patch is solved. There are two preset thresholds, a refinement threshold and a smaller coarsening threshold. The Physics code marks each facet of the spacetime mesh with an *adaptivity flag*, which can be either "refinement required", "coarsenable", or "neither". If the error estimate is below the coarsening threshold, the cell is marked as coarsenable. If the error estimate is above the refinement threshold, the cell is marked as "refinement required". Otherwise, the cell is marked as neither. Each cell in the space mesh also has an adaptivity flag, which is derived from the adaptivity flags on the spacetime cells as described below.

During each iteration, we do the following:

1. First, the Meshing module generates a patch. To generate a patch, we select a vertex with a local minimum time value. If any adaptivity operations are possible in the neighborhood of that vertex, we perform such an operation; otherwise, we pitch a tent at that vertex. Either option generates a patch.

2. Next, the patch is passed to the Physics module , which computes a solution and an a posteriori error estimate within each cell. If any error estimate is above the refinement threshold, the cell is marked as "refinement required" and the patch is rejected. Otherwise, the patch is accepted, and if any error estimate is below the coarsening threshold, the cell is marked as coarsenable.

3. If the patch is accepted, we store and output the new results and update the front. Also, for each outflow facet on the new front, we set its adaptivity flag to be equal to the adaptivity flag of its predecessor spacetime cell.

4. If the patch is rejected, we set the adaptivity flag of each *inflow* facet to match its successor in the patch. At least one such facet will be marked as "refinement required". We delete the patch, returning the spacetime mesh to its state before the patch was generated. We then refine all facets marked as "refinement required".

This workflow is similar to the workflow described by Abedi *et al.* [2] for $2d\times$time problems.

Unlike other algorithms that involve discrete remeshing operations, there is no projection error associated with translating the solution from the old space mesh to the new space mesh. This feature is a major advantage, because projection error can significantly decrease solution accuracy [16, 53, 13]. An adaptivity operation can only be performed if the newly generated simplices have positive volume and the outflow facets are adaptively valid. We attempt to perform edge flipping and smoothing only when such an operation would improve the worst quality of all the tetrahedra altered. We attempt to coarsen when the space cells that contain the vertex to remove are marked as coarsenable, and performing the coarsening would not create a tetrahedron with quality below a specific acceptable threshold. There is no theoretical guarantee that these operations are ever performed, but in practice, the conditions are not too difficult to satisfy.

After refinement and pitching, the resulting spacetime mesh is not necessarily a simplicial complex; a single predecessor cell may abut two or more successor cells. To handle this possibility, we change our data model slightly. We say that a simplex $A$ is a *subcell* of simplex $B$ if $A$ is the same dimension as $B$ and $A$ is

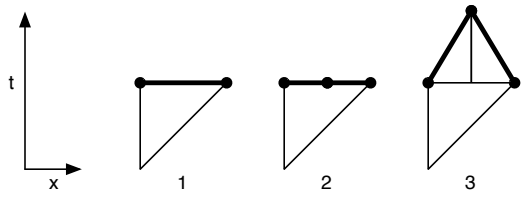


**Figure 3.10.** After refinement of the front (bold lines) shown in part (2), the next pitch (3) creates a spacetime mesh that is not a simplicial complex.

a subset of $B$. We say that $B$ is a *supercell* of $A$ if $A$ is a subcell of $B$. We define a *generalized simplicial complex* to be a set of simplices satisfying four conditions:

1. Faces of simplices in the complex are also in the complex.
2. The intersection of any two simplices in the complex is a subcell of a face of each simplex.
3. Given any two distinct cells of the same dimension whose intersection is also of that dimension, one is a subcell of the other.
4. Facets of the simplicial complex have no proper subcells.

When a cell $C$ on the front is refined, $C$ is no longer on the front, but $C$ is still a facet of its predecessors. Thus, the descendants of $C$ become subcells of $C$ in the new spacetime mesh. Since only cells on the front are refined, none of these cells are top-level cells in the spacetime mesh. By construction, the nesting property is also satisfied.

By construction, whenever a new spacetime cell is created, the intersection of it with any existing spacetime cell is either empty or a face of the new cell. Also by construction, this face of the new cell is a subcell of a face of the old cell. Thus, the intersection property is satisfied.

## 3.8 Adaptivity Examples

The figures below show the TentPitcher algorithm with both refinement and coarsening. This simulation started with a 1x1x1 cube mesh (6 tetrahedra). Since the physics code does not yet support the error calculations, we ran simulations using synthetic refinement/coarsening patterns. For the first simulation, we simulated a spherical wavefront that starts from a corner of the cube and moves outward at speed 0.8. Any cell whose bounding box crosses that wave is marked as refinable if the cell width is greater than 0.025. We mark a cell as coarsenable if the bounding box is entirely behind the spherical wavefront. Each time we pitch, we select a local minimum vertex $v$ and attempt the following adaptivity operations. First, we coarsen if all its neighbors are marked as coarsenable. Coarsening is performed by contracting $v$ to one of its neighbors. We do not coarsen if any tetrahedra would be generated with quality less than 0.001. Then, we attempt to flip away any of the edges (3-2 or 2-2 flip) or triangles (2-3 flip) that contain $v$ if the operation is legal and it would improve the worst quality tetrahedron among those altered.

The first image below shows the initial space mesh; the second image shows the space mesh after 100 patches have been accepted. At the beginning, there are a large number of rejections of patches and refinements of cells, and most of the refinement occurs in the corner where the wavefront begins. The third image

shows the space mesh after 3 million patches have been generated. The mesh in front of the wavefront gets finer closer to the wavefront, because whenever a cell intersecting the wavefront is refined the refinement can propagate to nearby cells. The mesh behind the wavefront has been coarsened. Farther behind the wavefront, more coarsening has occurred, because the cells in those areas have had more opportunities to be coarsened. Because of the aggressive coarsening, there are a lot of very high degree vertices close to the wavefront.

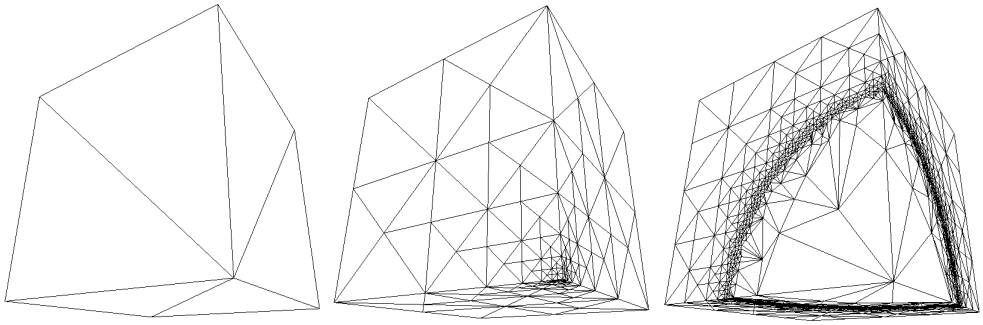

**Figure 3.11.** Cube mesh at the start, after 100 patches, and after 3 million patches.

Figure 3.12 shows multiple wavefronts crossing each other, providing situations in which regions of the mesh are refined, coarsened, and then refined again. In this simulation, two linear waves start from adjacent corners of the cube, and go outward on the front face at speed 0.8. The linear waves travel along the front face of the cube. Any cell whose bounding box intersects that wave is marked as refinable if the cell width is greater than 0.025. We mark a cell as coarsenable if either (1) its bounding box, projected onto the front face, is entirely behind at least one of the waves and is either behind or sufficiently far in front of the other wave, or (2) it is a distance of more than 0.1 away from the front face.

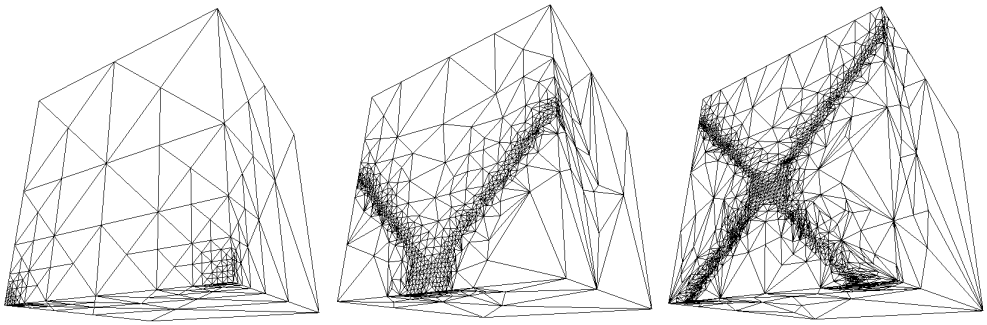

**Figure 3.12.** Cube mesh after 1,000 patches, after 300,000 patches, and after 1.5 million patches respectively.

These examples are not ideal, because the logic for deciding whether to refine or coarsen is not similar to that of the real physics data. In a real physics simulation, we will not know when solving a given cell whether it is in front of or behind the wavefront. To create these examples, we experimented with different parameters. Our objective was to stress-test the adaptivity algorithms and to create examples that clearly demonstrate many levels of refinement and coarsening rather than to accurately mimic the refinement and coarsening patterns of a real physics solution. Our experiments found that if we were to mark a cell as coarsenable whenever its bounding box does not touch the wavefront, we create situations in which a cell

touches the wavefront, is refined, and then one of the children is in front of the wavefront and thus is coarsened. This cycle of refinement and coarsening in front of the wavefront quickly degrades the quality of the mesh severely. By treating the front and back of the wavefront differently, we demonstrate the ability of our algorithm to perform very aggressive coarsening while avoiding this problem. A possible method of producing more realistic results is to say that a cell is marked as coarsenable if its bounding box, scaled up by a constant factor, intersects the wavefront. This method would avoid using any information about whether a cell is in front of or behind the wavefront.

# Chapter 4

# Visualization and Output

## 4.1 Visualization Capabilities

The visualization software has two modes: *space visualization*, which visualizes the state of the front at any given point in the execution of TentPitcher, and *spacetime visualization*, which visualizes the solution data at any given point in time. During space visualization, we visualize the state of the front after a given number of pitches, and the time value at each vertex is mapped to color using a cyclic color map. Thus, we can observe the colors change as the front advances. For spacetime visualization, we first run the output file through a *pre-renderer*, which converts the stream of patches into a series of simplices. The Physics code assigns each simplex a *group number*, which is used to determine which cells to visualize, and a set of scalar fields, each of which is represented by a polynomial. The *renderer* converts the output of the pre-renderer into a visualization. The visualizer takes in the following parameters:

1. A set of *visualization records*. Each visualization record gives a group to visualize, and whether to visualize it either by itself or as a facet of another group. If group $X$ is visualized as a facet of group $Y$, then for each cell of group $Y$ that has a facet that is in group $X$, that facet is visualized using the solution data from the cell in group $Y$ evaluated on that facet. (If two or more cofacets of the cell in group $X$ are of group $Y$, then that cell is visualized more than once.) For instance, suppose that we have a simulation consisting of a block of two materials separated by an interface, where the spacetime cells corresponding to the first material are marked as group 1, the cells corresponding to the second material are marked as group 2, and the cells on the interface are marked as group 3. By specifying that group 3 is to be visualized as a facet of group 1 and as a facet of group 2, we can visualize the values of the solution on both sides of the interface (these values can be different due to the discontinuity at the interface). Alternatively, a group can be visualized by itself; in that case, the solution data for each cell in that group comes from the cell itself.

   In the most common use case, the solution data is computed on the top-level spacetime cells, and we visualize $3d$ spacetime cells (since we display time slices of the cells visualized, each of these $3d$ spacetime cells generates a $2d$ time slice). Hence, for $3d\times$time simulations, we usually visualize cells on the boundary or at an interface, and we visualized $3d$ spacetime cells as facets of $4d$ cells. For $2d\times$time simulations, we usually visualize the entire solution, and cells are visualized by themselves.

2. An indication of which scalar field should be mapped to color and which scalar field should be mapped to height. The color field is applied to color the visualized surface, and the height field is applied to deform the surface in a specified direction, known as the *offset vector*. As described before, the

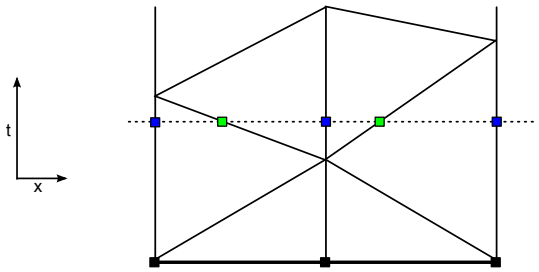


**Figure 4.1.** Example of slice mesh in $1d \times$time for clarity. Bold lines represent space mesh. Blue dots represent boundaries between 1-dimensional cells that are equivalent to those in the space mesh. Green dots represent boundaries that are created through intersection of the time slice with inflow and outflow facets.

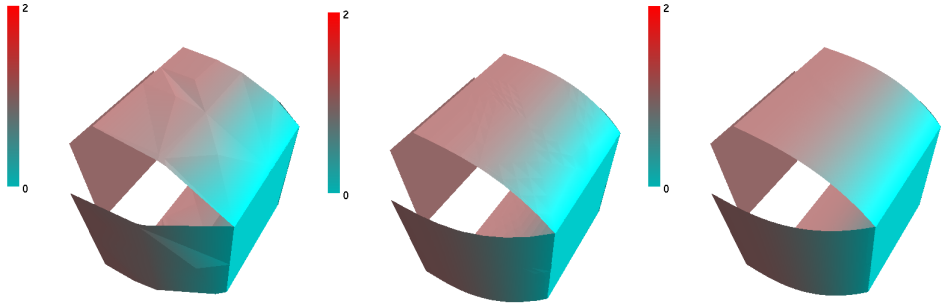


**Figure 4.2.** Example of a visualization of a synthetic cubic function mapped to both the height field and the color field, with each of the cube's six faces as a separate group. Shown are images of the rendering with each triangle not tessellated at all (left), and tessellated into 16 (middle) and 256 (right) smaller triangles.

polynomials that define this scalar field are in the output file from the pre-renderer. Details of how the offset vector is calculated are described below.

3. Which transfer functions to use to map the scalar fields to a point on the color map (for the color field) and an amount of displacement (height field) before they are visualized.

4. What time step to use between frames.

We render a frame at a given time value $t$ as follows. First, we compute the intersection of the spacetime mesh with the constant-time plane at time $t$. We refer to this intersection as the *slice mesh*. The boundaries between cells in the slice mesh include the boundaries of the original cells in the space mesh, as well as other boundaries where the inflow and outflow facets intersect the constant-time plane. For each visualization record, we compute offset vectors for each of the vertices contained in cells in that record. To compute the offset vectors for a vertex $v$ in a visualization record which has group $X$ visualized as a facet of group $Y$, we take the average, over all the 2-dimensional slices in the slice mesh of 3-dimensional cells of group $X$ that contain $v$, of the normal vectors of those cells in the direction pointing away from the cofacet of group $Y$.

Each triangle or quadrilateral to be visualized is itself tessellated into a number of smaller triangles for purposes of visualization. Each of these smaller triangles is rendered as a single primitive, with colors linearly interpolated across it. By using this tessellation, we can approximately render non-linear functions as curved surfaces or non-linear color fields.

## 4.2 Non-Adaptive Tent Pitcher

Figure 4.3 depicts the progress of the non-adaptive TentPitcher algorithm. Both pictures represent one run using a mesh of the Stanford bunny (a standard data set used to test computational geometry programs). We show two stages of the meshing process: after 1,000 and 50,000 pitching operations. After 1,000 pitching operations, not all the vertices have been pitched, so much of the mesh is still at the initial time value, indicated by a red color. After 50,000 pitching operations, the bunny has advanced non-uniformly in time.

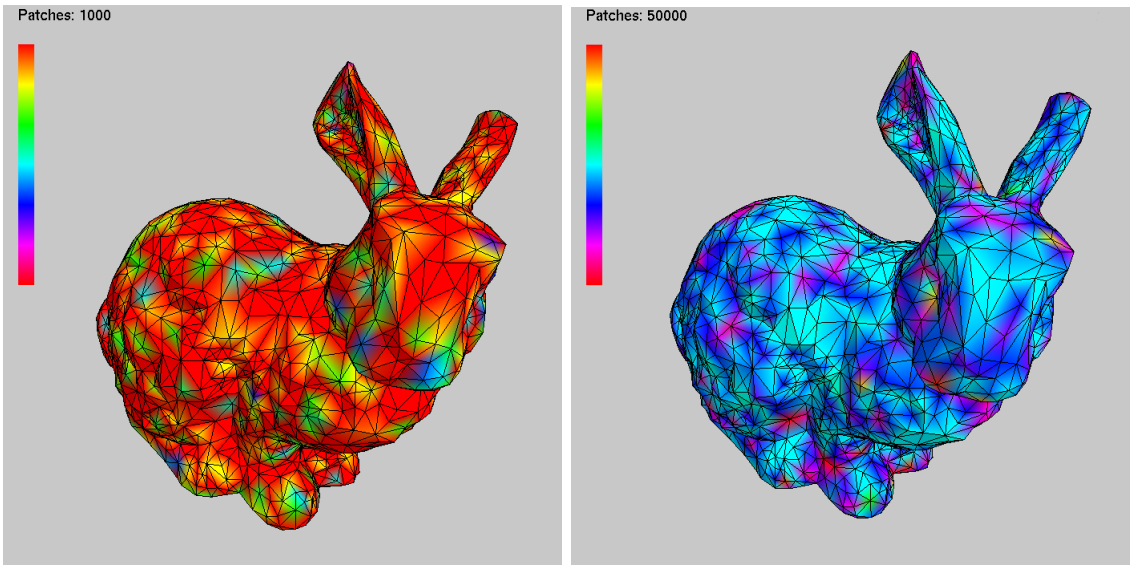


**Figure 4.3.** Stanford bunny after 1000 and 50000 patches.

## 4.3 Solution Data

As an example application, we combine our meshing code with unpublished physics simulation code being developed by Haber and Abedi. Consider a linear, elastodynamic simulation of wave scattering by a penny-shaped crack embedded in a solid cube that is subjected to sudden, tensile stress loading. The problem is solved in non-dimensional form using a $3d\times$ time extension of the spacetime discontinuous Galerkin method described by Abedi *et al.* [5] and Miller *et al.* [35]. The edges of the cube have length $L = 2$; the circular crack has radius $R = 0.25$, is centered in the cube, and is oriented parallel to two of the cube faces. The cube is comprised of an isotropic material described by Young's modulus, $E = 1$, mass density, $\rho = 1$, and Poisson ratio, $\nu = 0.3$. We apply symmetry boundary conditions to limit our simulation and visualization to one-eighth of the cube volume and a quarter-circle of the crack surface.

The cube is initially at rest and is unstrained. At time $t = 0$, we suddenly apply a uniform, unit surface traction to the two faces of the cube that are parallel to the plane of the crack and maintain this load throughout the simulation. One of the loaded faces includes the back face of the partial cube depicted in the visualizations shown in figures 4.4. We apply traction-free boundary conditions to the exterior cube faces and to the crack surface which appears at the lower-right corner of the front face in the figures. The visualizations show the log of the strain energy density mapped to color. The magnitude of the velocity field is mapped

to a height field constructed normal to the undeformed geometry on each face of the simulation model. The distinct normal values at corners and along edges of the simulation domain explain the discontinuous height field displays; the underlying deformation is, of course, continuous.

At early times we see the main wavefront due to the initial traction loading travel from front to back in the simulation domain. We observe scattering as the wave interacts with the crack, and we the high strain-energy values in the solid along the edge of the circular crack reflect the expected stress concentrations along the crack front. A complex pattern of scattered and reflected waves emerges as the simulation continues. Large velocities arise on the crack surface where the material motion is unconstrained.

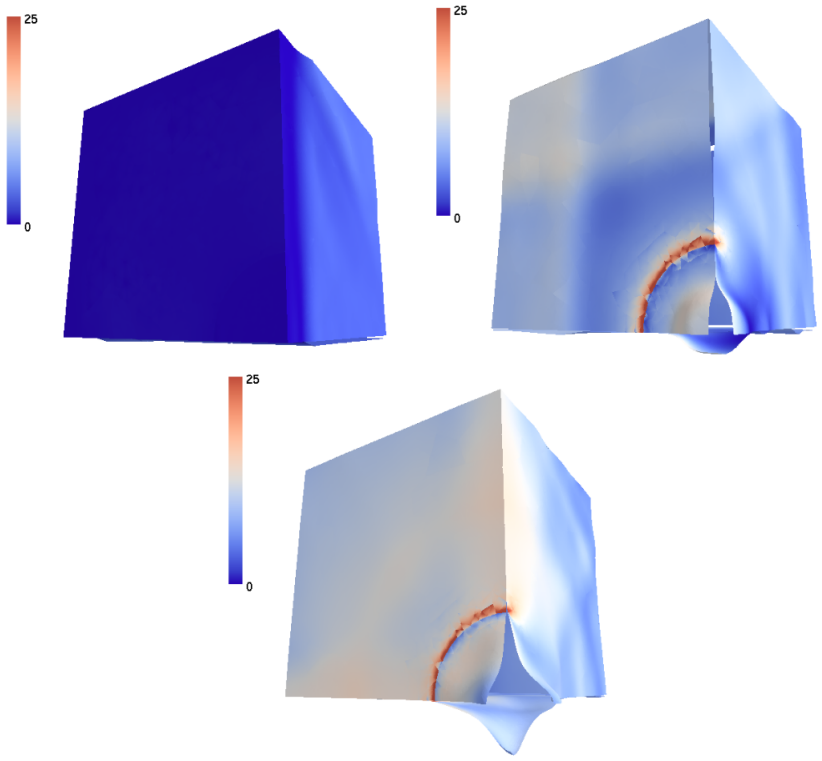


**Figure 4.4.** Penny-shaped crack simulation at 0.75, 1.5, and 2.25 seconds.

# References


[1] Adaptive remeshing for transient problems. *Computer Methods in Applied Mechanics and Engineering*, 75(1-3):195–214, 1989.

[2] Reza Abedi, Shuo-heng Chung, Jeff Erickson, Yong Fan, Michael Garland, Damrong Guoy, Robert Haber, John Sullivan, Shripad Thite, and Yuan Zhou. Spacetime meshing with adaptive refinement and coarsening. In *Proc. 20th Ann. Symp. Comput. Geom.*, pages 300–309, 2004.

[3] Reza Abedi, Shuo-heng Chung, Morgan A. Hawker, Jayandran Palaniappan, and Robert B. Haber. Modeling evolving discontinuities with space-time discontinuous Galerkin methods. In A. Combescure, R. de Borst, and T. Belytschko, editors, *IUTAM Symposium on Discretization Methods for Evolving Discontinuities*, volume 5 of *IUTAM Bookseries*, pages 59–87, 2007.

[4] Reza Abedi and Robert B. Haber. Spacetime dimensional analysis and self-similar solutions of linear elastodynamics and cohesive dynamic fracture. *Int. J. Solids and Structures*, 48:2076–2087, 2011.

[5] Reza Abedi, Robert B. Haber, and Boris Petracovici. A spacetime discontinuous Galerkin method for elastodynamics with element-level balance of linear momentum. *Computer Methods in Applied Mechanics and Engineering*, 195:3247–3273, 2006.

[6] Reza Abedi, Robert B. Haber, Shripad Thite, and Jeff Erickson. An $h$-adaptive spacetime-discontinuous Galerkin method for linearized elastodynamics. *European Journal of Computational Mechanics*, 15(6):619–642, 2006.

[7] Reza Abedi, Morgan A. Hawker, Robert B. Haber, and Karel Matouš. An adaptive spacetime discontinuous Galerkin method for cohesive models of elastodynamic fracture. *Int. J. Numerical Methods in Engineering*, 81:1207–1241, 2010.

[8] Reza Abedi, Boris Petracovici, and Robert B. Haber. A space-time discontinuous Galerkin method for linearized elastodynamics with element-wise momentum balance. *Computer Methods in Applied Mechanics and Engineering*, 195:3247–3273, 2006.

[9] Nina Amenta, Marshall Bern, and David Eppstein. Optimal point placement for mesh smoothing. *Journal of Algorithms*, 30:302–322.

[10] Douglas N. Arnold, Arup Mukherjee, and Luc Pouly. Locally adaptive tetrahedral meshes using bisection. *SIAM J. Sci. Comput.*, 22(2):431–448, 2001.

[11] Eberhard Bänsch. An adaptive finite-element strategy for the three-dimensional time-dependent Navier-Stokes equations. *J. Comput. Appl. Math.*, 36:3–28, 1991.

[12] Eberhard Bänsch. Local mesh refinement in 2 and 3 dimensions. *Impact of Computing in Science and Engineering*, 3:181–191, 1991.

[13] Marek Behr and Tayfun Tezduyar. Shear-slip mesh update in 3d computation of complex flow problems with rotating mechanical components. In *Computer Methods in Applied Mechanics and Engineering*, pages 3189–3200, 2000.

[14] James C. Caendish, David A. Field, and William H. Frey. An apporach to automatic three-dimensional finite element mesh generation. *International Journal for Numerical Methods in Engineering*, 21(2):329–347, 1985.

[15] P. Cignoni, D. Costanza, C. Montani, C. Rocchini, and R. Scopigno. Simplification of tetrahedral meshes with accurate error evaluation. In *IEEE Visualization 2000*, pages 85–92, 2000.

[16] Todd Dupont. Mesh modification for evolution equations. *Mathematics of Computation*, 39(159):85–107, 1982.

[17] Jeff Erickson, Damrong Guoy, John M. Sullivan, and Alper Üngör. Building space-time meshes over arbitrary spatial domains. *Eng. Comput.*, 20(4):342–353, 2005.

[18] David A. Field. Laplacian smoothing and Delaunay triangulations. *Communications in Applied Numerical Methods*, 4(6):709–712, 1988.

[19] Lori A. Freitag. On combining Laplacian and optimization-based mesh smoothing techniques. In *Trends in Unstructured Mesh Generation*, pages 37–43, 1997.

[20] Lori A. Freitag and Carl Ollivier-Gooch. Tetrahedral mesh improvement using swapping and smoothing. *International Journal for Numerical Methods in Engineering*, 40(21):3979–4002, 1997.

[21] P. J. Frey and P. L. George. *Mesh Generation: Application to finite elements*. Hermes Science, 2000.

[22] Glen A. Hansen, R.W. Douglass, and Andrew Zardecki. *Mesh Enhancement: Selected Elliptic Methods, Foundations, and Applications*. Imperial College Press, 2005.

[23] A. Jameson, W. Schmidt, and E. Turkel. Numerical solution of the Euler equations by finite volume methods using Runge Kutta time stepping schemes. In *AIAA 14th Fluid and Plasma Dynamics Conference*, 1981.

[24] Bryan Matthew Klingner and Jonathan Richard Shewchuk. Aggressive tetrahedral mesh improvement. In *Proc. 16th Int. Meshing Roundtable*, pages 3–23, 2007.

[25] Patrick M. Knupp. Matrix norms and the condition number: A general framework to improve mesh quality via node-movement. In *Eighth International Meshing Roundtable*, pages 13–22, 1999.

[26] Igor Kossaczký. A recursive approach to local mesh refinement in two and three dimensions. *J. Comput. Appl. Math.*, 55:275–288, 1994.

[27] Vladimir D. Liseikin. *Grid Generation Methods, 2nd Ed.* Hermes Science, 2010.

[28] Anwei Liu and Barry Joe. On the shape of tetrahedra from bisection. *Math. Comp.*, 63(207):141–154, 1994.

[29] Anwei Liu and Barry Joe. Quality local refinement of tetrahedral meshes based on bisection. *SIAM J. Sci. Comput.*, 16:1269–1291, 1995.

[30] R.B. Lowrie, P.L. Roe, and B. van Leer. Space-time methods for hyperbolic conservation laws. In *Barriers and Challenges in Computational Fluid Dynamics*, pages 79–98. Springer, 1998.

[31] Joseph M. Maubach. Local bisection refinement for $n$-simplicial grids generated by reflection. *SIAM J. Sci. Comput.*, 16:210–227, 1995.

[32] Joseph M. Maubach. The efficient location of neighbors for locally refined n-simplicial grids. In *Proc. 5th Int. Meshing Roundable*, pages 137–153, 1996.

[33] C.R.F. Maunder. *Algebraic Topology*. Courier Dover Publications, 1996.

[34] Scott T. Miller and Robert B. Haber. Space-time discontinuous Galerkin method for hyperbolic heat conduction. *Computer Methods in Applied Mechanics and Engineering*, 198:194–209, 2008.

[35] Scott T. Miller, Brent Kraczek, Robert B. Haber, and Duane D. Johnson. Multi-field spacetime discontinuous Galerkin methods for linearized elastodynamics. *Computer Methods in Applied Mechanics and Engineering*, 199:34–47, 2009.

[36] William F. Mitchell. *Unified multilevel adaptive finite element methods for elliptic problems.* Ph. D. thesis, Computer Science Department, University of Illinois, Urbana, IL, 1988. Tech. Rep. UIUCDCS-R-88-1436, http://math.nist.gov/~WMitchell/papers/thesis.pdf.

[37] William F. Mitchell. A comparison of adaptive refinement techniques for elliptic problems. *ACM Trans. Math. Soft*, 15:326–347, 1989.

[38] William F. Mitchell. Adaptive refinement for arbitrary finite-element spaces with hierarchical bases. *J. Comp. Appl. Math.*, 36:65–78, 1991.

[39] Doug Moore. Subdividing simplices. In David Kirk, editor, *Graphics Gems III*, chapter V.9, pages 244–249. Academic Press, 1992.

[40] James R. Munkres. *Elements of Algebraic Topology.* Perseus Books, 1993.

[41] Carl Ollivier-Gooch. Coarsening unstructured meshes by edge contraction. http://tetra.mech.ubc.ca/ANSLab/publications/coarsen.pdf.

[42] Steven J. Owen. A survey of unstructured mesh generation technology. In *Proceedings of the 7th International Meshing Roundtable*, pages 239–267, 1998.

[43] Jayandran Palaniappan, Robert B. Haber, and Robert L. Jerrard. A space-time discontinuous Galerkin method for scalar conservation laws. *Computer Methods in Applied Mechanics and Engineering*, 193:3607–3631, 2004.

[44] Jayandran Palaniappan, Scott T. Miller, and Robert B. Haber. Sub-cell shock capturing and space-time discontinuity tracking for nonlinear conservation laws. *Int. J. Numerical Methods in Fluids*, 57:1115–1135, 2008.

[45] V.N. Parthasarathy and Srinivas Kodiyalam. A constrained optimization approach to finite element mesh smoothing. *Finite Elements in Analysis and Design*, 9(4):309 – 320, 1991.

[46] Tomasz Plewa, Timur Linde, and V. Gregory Weirs. *Adaptive Mesh Refinement: Theory and Applications.* Springer, 2005.

[47] E. Granville Sewell. *Automatic generation of triangulations for piecewise polynomial approximation.* Ph. D. thesis, Department of Mathematics, Purdue University, West Lafayette, IN, 1972.

[48] Jonathan Richard Shewchuk. *Delaunay Refinement Mesh Generation.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[49] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computatioal Geometry: Theory and Applications*, 22(1–3):21–74, 2002.

[50] Hang Si. Tetgen: A quality tetrahedral mesh generator and three-dimensional Delaunay triangulator. http://tetgen.berlios.de/.

[51] Hang Si and Klaus Gärtner. Meshing piecewise linear complexes by constrained Delaunay tetrahedralizations. In *Proceedings of the 14th International Meshing Roundtable*, pages 147–163. Springer, 2005.

[52] Rob Stephenson. The completion of locally refined simplicial partitions created by bisection. *Math. Comp.*, 77:227–241, 2008.

[53] T.E. Tezduyar, M. Behr, and S. Mittal. A new strategy for finite element computations involving moving boundaries and interfaces - the deforming-spatial-domain/space-time procedure: II, computation of free-surface flows, two-liquid flows, and flows with drifting cylinders. *Computer Methods in Applied Mechanics and Engineering*, 94:353–371, 1992.

[54] Shripad Thite. *Spacetime Meshing for Discontinuous Galerkin Methods*. PhD thesis, University of Illinois at Urbana-Champaign, 2005. http://www.win.tue.nl/~sthite/pubs/phdthesis.pdf.

[55] Shripad Thite, Shuo-heng Chung, Jeff Erickson, and Robert Haber. Adaptive spacetime meshing in $2d\times$time for nonlinear and anisotropic media. In *8th US National Congress on Computational Mechanics, Minisymposium on Mesh and Geometry Generation*, 2005.

[56] B. H. V. Topping, J. Muylle, P. Ivanyi, R. Putanowicz, and B. Cheng. *Finite Element Mesh Generation*. Saxe-Coburg Publications, 2004.

[57] Christoph T. Traxler. An algorithm for adaptive mesh refinement in $n$ dimensions. *Computing*, 59:115–137, 1997.

[58] Alper Üngör and Alla Sheffer. Pitching tents in space-time: Mesh generation for discontinuous Galerkin method. *Int. J. Foundations of Computer Science*, 13(2):201–221, 2001.

[59] W. L. Wood. *Practical time-stepping schemes*. Clarendon Press, 1990.

[60] Yuan Zhou, Michael Garland, and Robert Haber. Pixel-exact rendering of spacetime finite element solutions. In *Proceedings of IEEE Visualization*, pages 425–432, 2004.

[61] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method: Fluid Dynamics, Volume 3*. Butterworth-Heinemann, 2000.

[62] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method: Solid Mechanics, Volume 2*. Butterworth-Heinemann, 2000.

[63] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method: The Basis, Volume 1*. Butterworth-Heinemann, 2000.